VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HRA V UNITY

GAME IN UNITY ENGINE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **RADEK UHLÍŘ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Uhlíř Radek**
Obor: Informační technologie
Téma: **Hra v Unity**
Game in Unity Engine
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte engine Unity.
2. Navrhněte inovativní hru v engine Unity.
3. Implementujte navrženou inovativní hru.
4. Proměřte a otestujte hru.
5. Vytvořte demonstrativní video.

Literatura:

- Dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a kostra aplikace

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Milet Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato bakalářka práce se zabývá navržením a implementací inovativní hry v engine Unity. Implementovat klasické herní mechaniky a vytvořit další ojedinělé herní mechaniky pro řešení různých logických hádanek. Práce se dále zabývá použitím nástrojů a komponent v editoru Unity a vytvořením dalších vlastních komponent.

Abstract

The main goal of this bachelor's thesis is to design and create innovative game in the Unity engine. Implement classic game mechanic and create more unique game mechanics for solving different logical puzzles. The thesis also focus on the use of tools and components within the Unity editor and creating custom components.

Klíčová slova

Unity engine, Unity editor, 3D hra, herní mechaniky, animace

Keywords

Unity engine, Unity editor, 3D game, game mechanics, animations

Citace

UHLÍŘ, RADEK. *Hra v Unity*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. TOMÁŠ MILET

Hra v Unity

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
RADEK UHLÍŘ
16. května 2018

Poděkování

Chtěl bych poděkovat vedoucímu mé bakalářské práce panu Ing. Tomáši Miletovi za odborné vedení, vstřícnost při konzultacích, věcné připomínky a za cenné rady při zpracování této práce.

Obsah

Kapitola 1

Úvod

Cílem práce bylo navrhnout a implementovat inovativní hru. Vymyslet a vytvořit herní mechaniky, díky kterým by se hra lišila od ostatních. Tvorbou her se každým dnem zabývá čím dál více lidí a firem. Skoro každý den vychází velké herní tituly a nespočetné množství her menších. Proto je potřeba tvořit hry, které jsou nějakým způsobem inovativní. Dnes už nestačí vytvořit libovolnou hru a doufat, že se nějak ujme na trhu. Je třeba vytvořit hry, které se liší od her ostatních. Protože proč by kdokoliv měl chtít zrovna moji hru, když má možnost si zahrát mnoho her jiných?

Tvorba inovativních herních mechanik je pouze jen část této práce. Abych mohl vytvářet inovativní mechaniky, nejdřív jsem musel vytvořit základní mechaniky a věci pro to potřebné. Proto se práce zabývá i tvorbou postav, pohybem hráče, animacemi, kolizemi, GUI a další.

1.1 Technologie pro tvorbu hry

Vývoj her je náročný proces vyžadující mnoho lidí. Hledaly se prostředky, jak tento proces maximálně zefektivnit a urychlit. Každá hra má stejný základ. Vyžaduje například render, detekci kolizí, přehrávání animací, umělou inteligenci, grafické uživatelské rozhraní nebo fyziku. Z tohoto důvodu vznikly herní engine [1], které všechny tyto základy řeší. Není je potřeba znovu vytvářet od začátku při tvorbě každé hry. V dnešní době existuje široký výběr mezi herními engine. Hru můžeme vytvořit například v Unreal Egine, Unity Egine či třeba Cry Engine. Jako vývojáři her bychom měli mít jasno, ve kterém herním engine hru vytvořit. Unreal Engine existuje už 20 let, obsahuje hodně nástrojů, obsahu a vytvořit jednoduchou hru se někdy dá i bez psaní kódu.

Pro tvorbu hry jsem zvolil egine Unity. Unity umožňuje vytvářet projekty na různé platformy. Nejen pro Windows a Linux, ale i na různé operační systémy chytrých telefonů, konzolí a jiných zařízení. Unity je zdarma pro nekomerční účely nebo pro firmy s obratem do 100 000 USD. Verze zdarma obsahuje většinu nástrojů verze placené. Kód je psán v jazyku CSharpScript [6] a je stejný jak na OS X tak Windows. Pro simulaci fyziky užívá engine PhysX [3] od NVIDIA. Unity Editor je snazší a přehlednější na používání než ostatní editory herních engine. K Unity existuje tisíce hodin video záznamů v podobě návodů, dokumentace [18] a manuál [19]. Unity má i dobře fungující fórum, kde lidé řeší problémy při tvorbě projektů. Díky široké komunitě pro editor Unity taktéž vzniká i hodně nástrojů (zdarma nebo placené), které jsou dostupné například v jejich vlastním obchodě (Unity Asset Store) [17].

Kapitola 2

Herní mechaniky

Základ každé hry jsou herní mechaniky. Schell [9, s. 41] popisuje herní mechaniky jako postupy a pravidla hry, které určují co, kdy a jak hráč může dělat, co je cílem a jak se lze k danému cíli dostat. Například herní mechanikou je, když hráč zmáčkne mezerník a postavička vyskočí na bednu. I prostor, ve kterém se hráč pohybuje a ve kterém je omezen je taktéž herní mechanikou. Inovativnost této hry spočívá v různých herních mechanikách, které nejsou u ostatních her. Hlavními dvěma inovativními mechanikami jsou schopnosti hráče:

- dočasně zpomalit čas,
- cestovat zpět v čase [4].

2.1 Zpomalení času

Jednou z nejdůležitějších herních mechanik je schopnost hráče zpomalit čas. Zpomalí pohyb všech objektů ve scéně s výjimkou sebe. Ke zpomalení času hráč používá svoji energii, které má omezené množství. Pokud je čas zpomalen, daná energie ubývá. Pohybuje-li se hráč při zpomaleném času, energie ubývá rychleji úměrně k hráčovi pohybu (otáčení, chůze a běh). Pokud čas není zpomalen, energie se hráči opět obnovuje. Zpomalení času dává hráči velkou možnost rozmyslet si následující pohyb, pečlivě plánovat a nepanikařit. Zpomalení času je důležitou schopností hráče pro pohyb v některých scénách a vyřešení mnoha hádanek.

2.1.1 Příklady použití

- Překonání propasti přes starý dřevěný most. Jakmile hráč na most stoupne, most hráčovu váhu neudrží, most se zřítí a hráč spadne do propasti. Ale zpomalí-li hráč čas, může přeběhnout po částech mostu, než se most do propasti zřítí.
- Hráč se potká s nepřítelem, který po hráči vystřelí ze zbraně. Hráč v ten moment zpomalí čas a může vidět zpomalenou kulku letící na něj. Hráč tak může odhadnout kam uhnout, aby ho kulka nezasáhla, přičemž může i odhadnou kam vystřelit, aby nepřítele zasáhl.
- Hráč se opět potká s nepřítelem, ale tentokrát má nepřítel v ruce třeba meč. Nepřítel se napřáhne a pokusí se do hráče mečem seknout. Hráč opět zpomalí čas a může se vyhnout úderu od nepřítele. Taktéž hráč může nepřítele udeřit třeba do ruky, čímž

nepřítel zbraň držící v ruce upustí, sebrat zbraň nepříteli a nepřítele jeho vlastní zbraní zneškodnit ve zpomaleném čase.

2.2 Cestování do minulosti

Další velmi důležitou a výrazně složitější herní mechanikou je super schopnost hráče cestovat zpět v čase. Během hraní hry hráč musí nejdříve vytvořit bod v čase, do kterého se bude chtít později vrátit. V momentě vytvoření bodu v čase se uloží stav celé scény a spustí se nahrávání všech hráčových činností (kudy šel, vyskočil, sebral, použil nebo třeba odhodil předmět). V momentě, kdy se hráč rozhodne cestovat zpět do bodu v minulosti (vrátit se v čase), ukončí se nahrávání událostí a celá scéna se vrátí do stavu, ve kterém byla, než hráč provedl vytvoření daného bodu v čase. Jelikož se ale jedná o cestování v čase a nejen o vrácení stavu hry do určitého bodu, hráč je najednou ve scéně dvakrát. Jeho minulé já (dále jen jako „Duch“), které vytvořilo bod v čase a jeho současné já (dále jen jako hráč), které cestovalo z budoucnosti do minulosti. Při vrácení se v čase se spustí přehrávání nahraných událostí. To znamená, že Duch (hráčovo minulé já) postupně provede všechny akce v pořadí, které předtím provedl hráč (v průběhu času Duch jde tam kam hráč šel a provede akce, které hráč provedl). Hráč mezitím může svoji postavičku bez žádného omezení dál ovládat. Jakmile skončí přehrávání všech událostí, Duch se vrátí v čase (zmizí ze scény).

2.2.1 Předmět z minulosti

Když hráč cestuje z budoucnosti do minulosti, může mít u sebe v ruce předmět. Tento předmět si hráč vezme s sebou zpět do minulosti. To znamená, že stejně jak postava hráče bude ve scéně dvakrát, tak teď i daný předmět bude ve scéně dvakrát. Předmět, který byl ve scéně v momentu vytvoření bodu v čase (dále jen jako „Originál“) a teď i druhý předmět, který si hráč vzal s sebou z budoucnosti do minulosti (dále jen jako „Kopie“).

Kopie se nijak neliší od Originálu. Jedná se o dva identické předměty jen s tím rozdílem, že Kopie se vloží do inventáře hráče a Originál se vrátí do stavu, ve kterém byl při vytvoření bodu v čase. Pokud byl Originál na zemi, je zpátky na zemi, pokud byl u nepřítele v inventáři, bude opět v daném inventáři. Pokud byl Originál u hráče v inventáři při spuštění nahrávání, je přemístěn k Duchovi do inventáře. Musíme si pořád uvědomovat, že Duch je hráčovo minulé já a při vytvoření bodu v čase byl předmět u hráče (z našeho pohledu po vrácení se v čase to znamená u Ducha).

Během přehrávání událostí, můžou Duch a hráč spolu interagovat. Duch postupně provádí všechny akce, které předtím provedl hráč. Například, pokud hráč někam šel, sebral a použil tam předmět, i Duch ho teď taky sebere a použije. Ale pokud hráč Ducha předběhne, daný předmět sebere dřív než Duch a nechá na stejném místě předmět jiný, tak Duch až na to místo přijde, sebere a použije ten předmět, který tam hráč nechal. Například, pokud hráč původně při nahrávání sebral meč, použil meč a po vrácení se v čase hráč na místo meče umístí třeba střelnou zbraň, Duch sebere zbraň a vystřelí z ní.

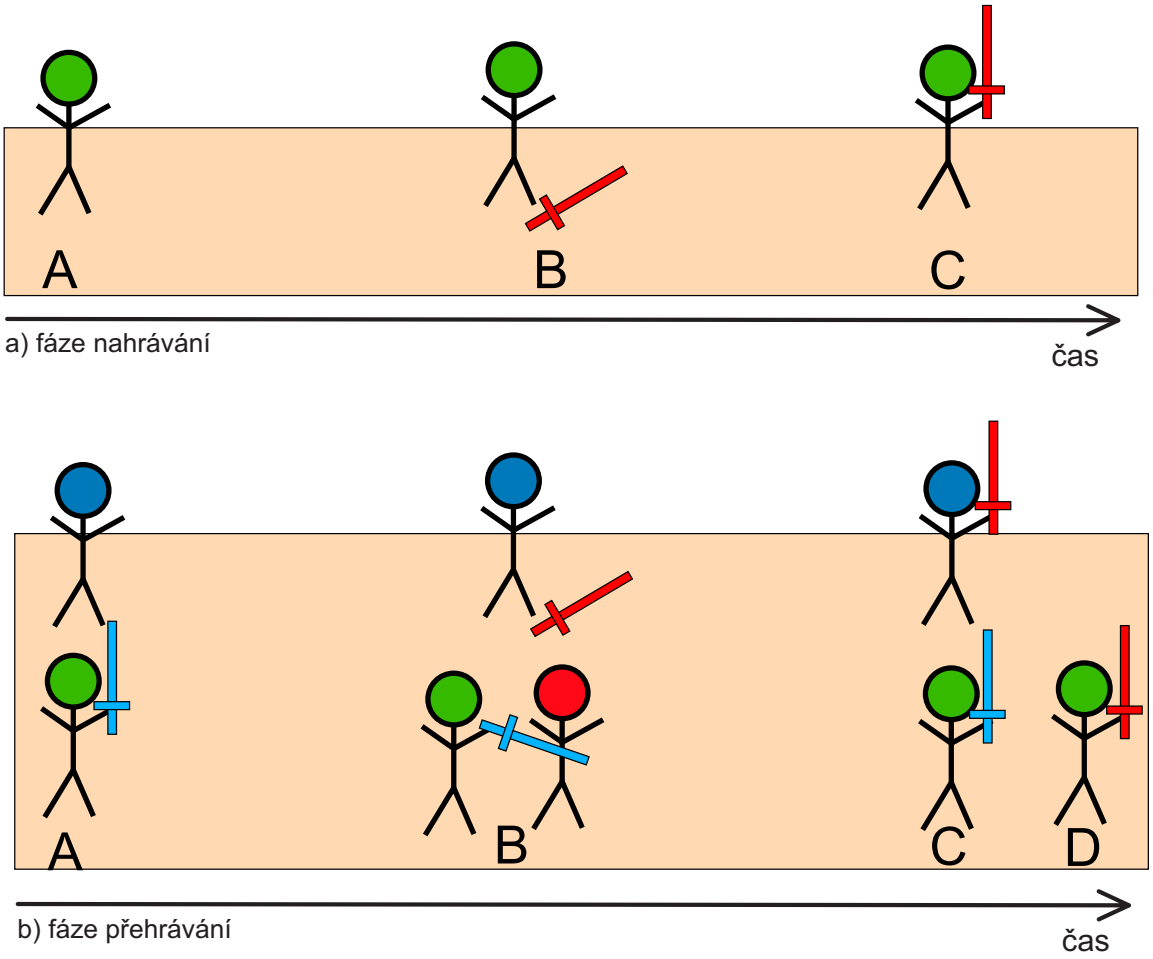
Víme tedy, že na konci přehrávání, když Duch se vrátí v čase (zmíní ze scény), může mít v ruce jiný předmět, než měl hráč, když se hráč vracel v čase. Proto musí být zajištěna konzistence předmětů ve scéně. Kopie, která se vytvořila, když hráč se vrátil v čase, musí být odstraněna, když se Duch vrátí v čase. Na místo odstraněné Kopie je vždy přemístěn Originál. Po skončení přehrávání jsou ve scéně stejné předměty jako před začátkem nahrávání.

Po cestování Ducha v čase nastane jedna z akcí pro zachování časového kontinua:

- Pokud hráč cestoval v čase s předmětem a Duch se vrátí v čase s Originálem, Originál dostane do ruky hráč.
- Pokud hráč cestoval v čase bez předmětu a Duch se vrátil s předmětem, předmět se přesune hráčovi do ruky. Měl-li hráč v době vrácení se Ducha do minulosti v ruce již jiný předmět, onen jiný předmět hráč odhodí.
- Pokud hráč cestoval v čase s předmětem, ale Duch se teď vrátil bez předmětu, tam kde byla Kopie nebude žádný předmět.
- Pokud hráč cestoval v čase s předmětem, ale duch se vrátil s předmětem jiným (Kopie je například u hráče v inventáři), předmět Ducha se vloží hráčovi do inventáře.
- Vrátí-li se duch v čase s Kopií místo Originálu, není potřeba provádět akci (Kopie zmizí ze scény spolu s Duchem a zůstane ve scéně Originál).
- Pokud hráč cestoval v čase bez předmětu a Duch se vrátí v čase taktéž bez předmětu, není třeba provádět akci.

Znamená to tedy, že pokud změníme minulost, změní se i naše přítomnost. Když si tento fakt uvědomíme, tak víme, že musíme plánovat nejen prostorem a časem do budoucna, ale musíme i plánovat, co se stane v minulosti. Plánovat do minulosti je pro většinu nezvyklé, a proto z počátku velmi náročné na řešení hádanek. Je to herní mechanika, o které si dovoluji říct, že v žádné jiné hře není.

Herní mechanika cestování zpět v čase může být těžší na pochopení, na obrázku 2.1 je příklad pro lepší vysvětlení.



Obrázek 2.1: Na obrázku jsou tři typy postav. Hráč zeleně, Duch modře a nepřítel červeně. Červený meč je sebratelný předmět (Originál), modře je dočasná kopie červeného meče (Originálu). Obrázek je rozdělen do dvou částí:

(a) fáze nahrávání:

- V čase A hráč spustí nahrávání. Meč je na zemi. Uloží se stav celé scény.
- V čase B hráč sebere meč. Zaznamená se událost sebrání předmětu.
- V čase C se hráč vrátí v čase s mečem v ruce. Ukončí se fáze nahrávání a spustí se fáze přehrávání.

(b) fáze přehrávání:
Spuštěním fáze přehrávání se hráč dostane zpět na místo, kde byl, když spustil nahrávání a bude ve stavu v jakém byl na konci nahrávání. Na stejném místě, kde je hráč se objeví i Duch (hráčovo minulé já). Duch bude ve stejném stavu, v jakém byl hráč, když spustil nahrávání. Hráč si z budoucnosti donesl meč (Originál), do inventáře dostane Kopii. Originál se společně s ostatními objekty vrátí do stavu ve kterém byly, když bylo spuštěno nahrávání (meč je opět na zemi). Duch začne opakovat akce, které byly ve fázi nahrávání provedeny hráčem.

- V čase A hráč i Duch jsou na stejném místě. Duch má prázdný inventář, hráč má u sebe meč (Kopii). Originál je na zemi.
- V čase B Duch provede zaznamenanou událost – sebere meč (Originál). Mezitím ve stejném čase hráč může provádět jiné akce, např. bojovat s nepřítelem.
- V čase C se Duch vrátí v čase a aktualizuje se časové kontinuum. Takže v čase D má hráč v inventáři Originál.

Kapitola 3

Návrh a implementace

V této kapitole popisuji jednotlivé funkcionality a části hry. Dále popisuji nástroje v editoru Unity, které jsem použil na tvorbu scén a postav. Zaměřil jsem se i na různé výhody, nevýhody a problémy engine a jak tyto problém řešit.

3.1 Scény

Projekty v Unity se skládají z jednotlivých scén. Scéna je prostor obsahující objekty, zdroje zvuků, světla, kamery, efekty atd. Tato hra se skládá z 10 scén (hlavní menu a 9 herních scén mezi kterými hráč přechází). Jednotlivé herní scény demonstrují různé mechaniky. Jednu herní scénu lze chápat jako jednu unikátní úroveň. Většina geometrie scén byla tvoře přímo v editoru Unity pomocí nástroje ProBuilder [2]. Ve všech scénách byla vygenerována světelná mapa s ambient occlusion. V každé scéně bylo použito jedno přímé světlo bez stínů a několik světel bodových s měkkými stíny.

Hlavní menu

Scéna Hlavní menu neobsahuje hráče ani žádné herní mechaniky. Tato scéna slouží jen jako počáteční scéna, která se má načíst, když uživatel spustí hru. Z grafického hlediska obsahuje jen menu. Menu je taktéž dostupné v herních scénách.

Úroveň první – prosté zpomalení času

Účel první scény je, aby si hráč zvykl na pohyb postavy, uvědomil a odzkoušel si speciální schopnost postavy zpomalit čas. Jedná se o nenáročnou úroveň na myšlení. Bez zpomalení času nelze zvládnout. Scéna obsahuje tři překážky:

- Starý most: hráč musí zpomalit čas, aby mohl přejít po částech mostu, než se celý most zřítí.
- Bedny nad propastí: když hráč zpomalí čas, jeho tělo minimálně ovlivňuje dynamické objekty. To mu dává možnost shodit bedny do propasti, zpomalit čas a pak přes bedny, které jsou ve vzduchu, přeskočit na druhou stranu.
- Nestabilní plošiny: mezi hráčem a koncem úrovně je propast nad kterou jsou plošiny. Avšak plošiny neudrží hráčovu váhu. Hráč musí ve zpomaleném čase přes plošiny přeskákat do cíle.

Úroveň druhá – interaktivní prvky a zpomalení času

Jedná se o další ještě relativně nenáročnou scénu, kde se hráč poprvé setká s interaktivními prvky v kombinaci se zpomalením času.

- Dvojice dveří: první dveře hráč otevře stiskem tlačítka ve scéně a zůstanou otevřeny. Druhé dveře se taktéž otevřou stiskem tlačítka, ale vzápětí se zavřou. Hráč musí zpomalit čas a projít dveřmi, než se opět zavřou.
- Jednoduché zapnutí laseru: hráč se poprvé setká s předmětem bedna, kterou sebere a položí ji na platformu pro zapnutí laseru. Laser vzápětí otevře dveře.
- Předběhnutí laseru: prostřednictvím tlačítka ve scéně hráč zapne laser, ale laser začne blokovat cestu ke dveřím. Hráč musí zpomalit čas, předběhnout laser, než laser zablokuje cestu a hráč nebude moct projít.
- Odražení laseru: před hráčem jsou tři zrcadla, která slouží pro odražení laseru. Dvě jsou statická (obr. 3.2), ale mezi nimi je jedno, které lze sebrat do inventáře (obr. 3.5). Hráč položí ve scéně zrcadlo tak, aby odchýlil laser směrem ke dveřím.

Úroveň třetí – cestování zpět v čase

Hráč se v této scéně alespoň částečně musí naučit, jak herní mechanika cestování zpět v čase funguje. V každé části scény má vždy k dispozici jen jednu bednu a musí vymyslet, jak ji použít. Tato úroveň už vyžaduje hlubší zamyšlení, jak herní mechanika funguje.

- Polož bednu, otevřou se ti dveře: hráč má vedle sebe k dispozici bednu, kterou když sebere a položí na platformu před dveřmi, dveře se otevřou. Jednoduchý princip, který ukazuje, čeho bude muset ve scéně opakovaně dosáhnout různými způsoby, aby se dostal dál.
- Časový paradox [4]: hráč se dostane do zóny, kde má dovoleno cestovat zpět v čase. Musí projít dveřmi, které ale nelze nijak otevřít. Před dveřmi je platforma na bednu, ale žádná bedna kolem, kterou by hráč mohl sebrat. Ale jedna bedna leží za zavřenými dveřmi. K této bedně se však hráč nemůže normálně dostat. Hráč musí vytvořit časový paradox, ve kterém projde zavřenými dveřmi a vrátí se s bednou zpět v čase. Tím získá bednu, kterou může použít k otevření dveří, které byly zavřeny, když hráč nimi procházel. Následně Duch i hráč může danými dveřmi projít.
- Klonování předmětů: pro otevření dveří potřebuje hráč bedny dvě, ale k dispozici má bednu jen jednu. Hráč se vrátí s bednou v čase, čímž bude mít dočasně k dispozici potřebné bedny pro otevření dveří.
- Pomocí vrácení se v čase hráč musí přijít na to, jak pronést bednu přes dveře.

Úroveň čtvrtá – deska a Duch

V předešlých úrovních se hráč seznámil se základy herními mechanikami. Zde je musí využít:

- Pro předmět deska nechoď: ukázka, jak získat předmět, bez toho, aby pro předmět musel chodit.

- Spolupráce Ducha a hráče: Duch vynese hráče na vyvýšené místo, kam by se hráč jinak nedostal.
- Hoď Desku a zpomal čas: hráč se musí dostat z jednoho vyvýšeného místa na druhé.

Úroveň pátá – lasery

Všechny předcházející scény postupně ukazovaly části některých herních mechanik krok za krokem. Hráč problém vyřešil a šel dál. V této scéně (obr. 3.3 a 3.4) už však není zřejmé, co by měl hráč asi udělat. Hráč si scénu musí projít, seznámit se se všemi jejími částmi a zkoušet co má udělat. Taktéž musí využít mechaniky zpomalení času a cestování zpět v čase opakovaně.

Úroveň šestá – okno

Každá z úrovní byla na myšlení více a více obtížnější. Tato už je skutečný hlavolam. Hráč musí pomocí cestování zpět v čase klonovat předmět a zároveň s Duchem opakovaně spolupracovat (podávat si předmět přes okno a vzájemně si různými způsoby otevírat dveře).

Úroveň sedmá a osmá – nepřátelské AI

V těchto scénách si hráč mírně odpočine od hádanek, zde si naopak zabojuje. Po načtení úrovně je zpomalen čas a před hráčem je nepřátelský voják, který chce hráče zabít. Část scény osm je na obrázku 3.22. Hráč může zaútočit na nepřítele – udeřit ho rukou. Zraněný nepřítel upustí zbraň, hráč může zbraň nepříteli sebrat a nepřítele jeho vlastní zbraní zabit. Hráč se následně musí probojovat až na konec scény.

Úroveň devátá

Prozatím poslední scéna, kde hráč musí vyřešit různé hádanky a zároveň přemoct nepřítele. Jedná se o další scénu, která kombinuje jednotlivé herní mechaniky. Hráč opět musí zpomalovat čas, cestovat zpět v čase, a hlavně spolupracovat s Duchem. Hráč musí například hodit předmět Duchovi, ten ho musí použít a zároveň hráč musí ochránit Ducha před nepřítelem.

Čím složitější je úroveň, tím v ní existuje více způsobů, jak vyřešit její hádanky. Tím nemyslím boj s nepřítelem, třeba kterého vojáka zneškodnit dřív. Čím je úroveň větší a delší, kde hráč je nucen kombinovat herní mechaniky, existuje i více způsobů, jak hádanky ve scénách vyřešit. Například scény pátá, šestá a devátá, kde hráč musí cestovat zpět v čase, lze vyřešit různými způsoby. Může vytvořit větší časový paradox nebo naopak několik menších jednodušších časových paradoxů.

3.2 Inventář a předměty

Z hráčova pohledu inventář podporuje 4 akce – sebrat, použít, upustit a zahodit předmět. Provedení akce trvá nenulový čas. Hráč může mít v inventáři současně pouze jeden předmět. Pokud v inventáři je předmět a hráč se pokusí jiný sebrat, tak aktuální předmět upustí a o moment později druhý předmět sebere. I když to není z hráčského pohledu znát, hráč ve skutečnosti může nosit předměty dva. Předmět, který lze sebrat a ruku, která je ve skutečnosti taky předmětem. Hra obsahuje celkem šest předmětů (3 zbraně a 3 další předměty pro řešení hádanek).

3.2.1 Sebratelné předměty

Každý předmět, který lze vložit do inventáře, se skládá vždy alespoň ze třech částí: správce předmětu, předmět do ruky a předmět ve scéně. Správce předmětu spravuje předmět a řídí například aktivaci či deaktivaci ostatních částí předmětu. Vždy je buď aktivní předmět ve scéně a je vypnutý předmět do ruky nebo přesně naopak. Důležitým základem je si uvědomit, že předměty, které můžeme ve scéně sebrat, jsou pouze reprezentací daného předmětu a samy o sobě nemají žádnou vlastnost pro hráče. V momentě, kdy hráč předmět sebere, tak se předmět ve scéně deaktivuje a předmět do ruky aktivuje. Předmět do ruky a předmět ve scéně nemají společné vlastnosti a každý se chová úplně jinak. Jedinou část, co některé části předmětu mají společné je model, který se vykresluje.

Ve scéně jsou interaktivní body (komponenta *InteractionPoint*), které jsou přidány na různé objekty. Například na předmětech do inventáře, tlačítkách či na jakémkoliv objektu, se kterým chce hráč manipulovat. Na hráčovi je komponenta *EasyPickUp*. Když se hráč přiblíží k interaktivním bodům a chce s některým manipulovat (sebrat předmět, otočit zrcadlo či zmáčknout tlačítko), hráč se dotáže komponenty *EasyPickUp* a komponenta vyhodnotí, se kterým interakční bodem chtěl hráč asi manipulovat. Vyhodnocení, který interakční bod se vybere má dvě fáze:

- Test viditelnosti: komponenta *EasyPickUp* zkontroluje, zda interakční bod je vidět.
- Minimální úhel: z viditelných objektů se vybere interakční bod, který má nejmenší úhel k hráči.

Jednoduchý test na viditelnost objektu je ve výpisu 3.1. Test nezjišťuje, zad objekt je skutečně vidět, ale zda mezi bodem pozorovatele a cílem neexistuje překážka. Když překážka neexistuje, předpokládá se, že objekt je vidět. Test je v podobě paprsku vytvořeného zavoláním statické metody *Physics.Linecast*.

Předmět ve scéně

Každý předmět ve scéně obsahuje komponenty *Rigidbody* a *Collider* pro simulaci fyziky, interakční bod, komponentu *PickUpObject* reprezentující předmět ve scéně a model předmětu.

Předmět do ruky

Každý předmět do ruky je odvozen z bázové třídy *InHandObject* a jejich vnitřní implementace se různě liší. Předmět v ruce nemá *Rigidbody* a taktéž ani *Collider*.

Předměty:

- Ruka: každá postava má tento předmět u sebe v inventáři. Je to jediný neupustitelný předmět. Je to předmět, který hráčovi jako předmět nepřipadá, neobsahuje model pro vykreslení. Reprezentuje ho část postavy.
- Meč: předmět stejného typu jako ruka, ale s jinými parametry. Má např. větší dosah účinnosti než ruka a jiná časování. Vhodná zbraň pro boj s nepřítelem z blízka.
- Střelná zbraň: použitím tohoto předmětu se provede výstřel a vytvoří se kulka. Vhodná zbraň pro boj s nepřáteli na dálku.

- Modrá bedna: slouží jako energii či klíč pro aktivaci objektů (laserů a dveří).
- Deska: hráč a Duch mohou použít tento předmět pro přenos dynamických objektů z bodu A do B. Hlavní použití předmětu je, aby Duch přenesl hráče (hráč si vyskočí na Ducha) a hráč se může dostat na vyvýšené místo.
- Stojan se zrcadlem: pro přesměrování laserů.

```
1   protected override bool checkVisibility(Transform target) {
2
3       RaycastHit hit; // Struktura, do které se ukládají informace o překážce
4
5     /* lockingLayers je 32--bitová maska; vrstvy, přes které nelze předmět sebrat
6     (např. stěna na vrstvě Default) */
7       if (Physics.Linecast(pickerPivot.position, target.position, out hit,
            blockingLayers)) {
8         // mezi předmětem a hráčovou hlavou je překážka (collider)
9           return false;
10      } else {
11      // mezi předmětem a hráčovou hlavou není překážka, předmět lze sebrat
12          return true;
13      }
14  }
```

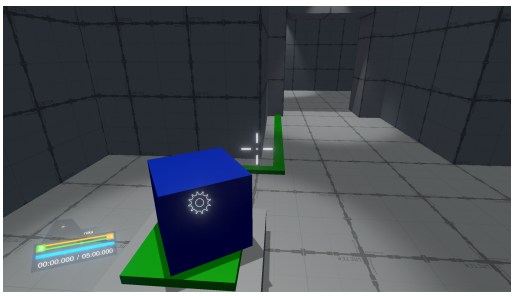
Výpis 3.1: Test na viditelnost objektů pomocí paprsku mezi dvěma body.

3.2.2 Ostatní interaktivní objekty

Předměty do inventáře nejsou jediné objekty, se kterými hráč může interagovat.

Platforma pro bednu

Hráč může položit předmět bednu na platformu. V důsledku kolize bedny s platformou se platforma aktivuje, přepne se její barva z červené na zelenou a provede akci platformě přiřazenou (např. zapnout laser či otevřít dveře). Hráč bednu opět může sebrat nebo shodit dolů, čímž platforma přiřazené zařízení opět vypne. Příklad platformy s bednou je na obr. 3.1



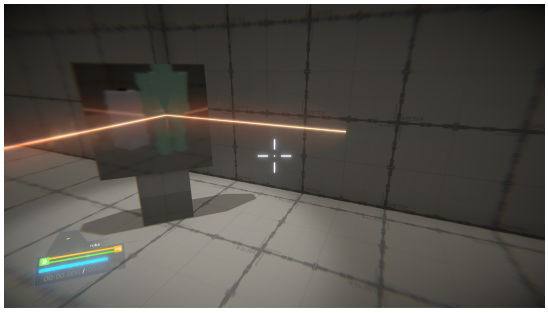
Obrázek 3.1: Modrá bedna na platformě otevřela dveře.

Laser

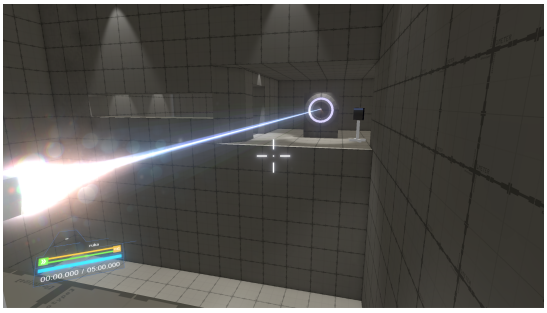
Laser v této hře nemá nekonečnou rychlost, jak je tomu v jiných hrách. Když se laser aktivuje, tak trvá, než se laser dostane do koncového bodu. Ve hře existují dva typy laseru. Liší se v barvě a rychlosti. Oranžový, který je pomalejší a při zpomalení času lze předběhnout (obr. 3.2 a 3.5). A bílo-modrý, který je výrazně rychlejší a nelze ho předběhnout. Při manipulaci s laserem hráč musí být opatrný, protože se laserem může zabít, popřípadě může laserem zabít nepřítele.

Když se laser aktivuje, tak má nulovou délku. V průběhu času jeho koncový bod putuje prostorem, dokud nenarazí na překážku. Překážkou může být libovolný objekt ve scéně. Laser rozeznává tři typy překážek:

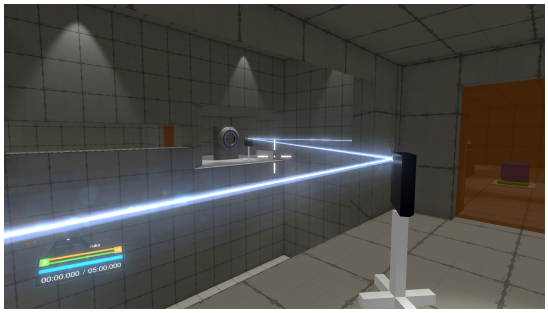
- Zrcadlo: určuje nový směr laseru podle normály strany modelu v komponentě *Collider* na překážce, laser následně dál hledá cestu v prostoru.
- Zařízení pro příjem laseru: přijímá energii od laseru, aktivuje jiné zařízení a neodráží laser.
- Jiné: o jakýkoliv jiný typ objektů se laser zastaví a dál nepokračuje.



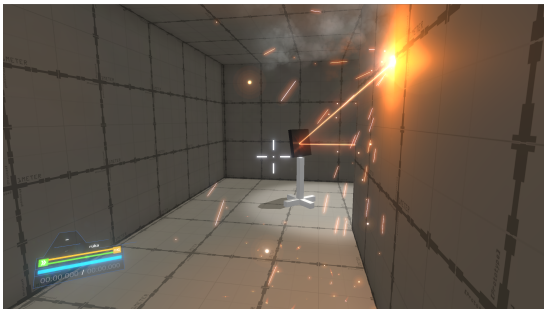
Obrázek 3.2: Laser odražený od statického zrcadla.



Obrázek 3.3: Laser napájí zařízení na druhé straně propasti.



Obrázek 3.4: Laser je opakovaně odražen pomocí stojanů se zrcadly.

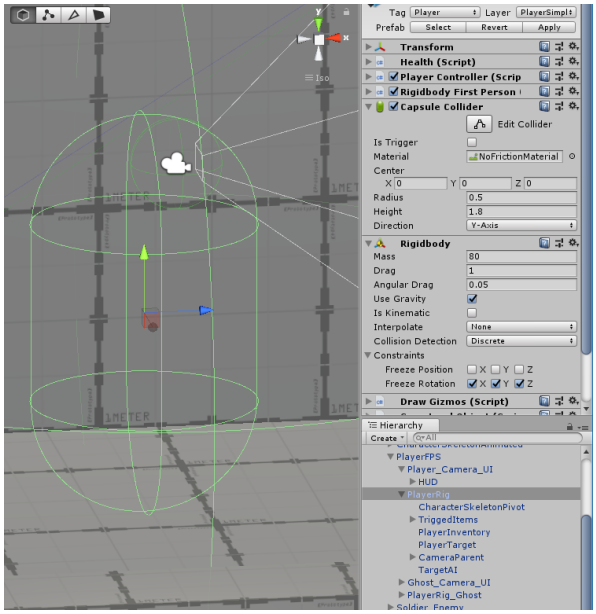


Obrázek 3.5: Laser je odražen zrcadlem do stěny, kde laser končí.

3.3 Postava hráče

V Unity je simulace fyziky všech objektů založen na fyzikálním engine PhysX [3]. Hráč se pohybuje za pomocí komponent *Rigidbody* a *Capsule Collider* (kolizní model). Komponenta *Rigidbody* slouží k simulaci fyziky a je základem každého objektu ve scéně, na kterém chceme simulovat fyzikální chování. Ať už hození předmětu, kutálející se kuličky, padající kámen nebo bednu, do které narazíme a posune se. Bez přidání jakéhokoliv dalšího kódu, každý objekt s *Rigidbody* je přitahován k zemi gravitací a v případě protnutí objektu s jiným objektem, kde alespoň jeden z nich má *Rigidbody*, bude detekována kolize. Předmět bez *Rigidbody* se nehýbe. Pokud by na hráčovi nebyl přidán *CalpsuleCollider*, nereagoval by s prostředím a propadl by zemí. Unity nám umožňuje nastavit i sílu tření a odrazu při kolizi

s jiným objektem prostřednictvím fyzikálních materiálů, který můžeme přiřadit jakékoliv komponentě typu *Collider*.



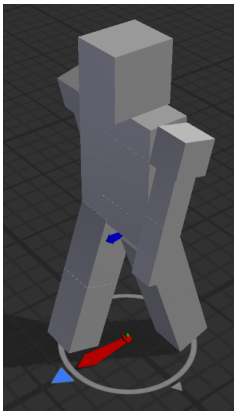
Obrázek 3.6: Objekt hráč vybrán v editoru Unity.

Na hráče jsem použil fyzikální materiál s nulovým třením, a to z důvodu, aby nebyl zpomalován, když jde například podél zdi. Pokud by měl například třecí sílu rovnu jedné a jen by se dotkl třeba zdi, okamžitě by se zastavil a podél zdi by se nemohl vůbec hýbat.

Na obrázku 3.6 je zobrazena komponenta *Capsule Collider*, která je vykreslován v editoru zeleně. Přes *Inspector* komponentu *Rigidbody* a jiné komponenty. Taktéž je vidět, že v hierarchii hráče je např. inventář a kamera, ale nikde není vidět 3D postava hráče. Pohyb (fyzika) hráče je nezávislý na postavě, která má být vykreslována. Místo postavy, která byla použita, může být použita postava o jiné velikosti a na fyziku postavy to nebude mít vliv.

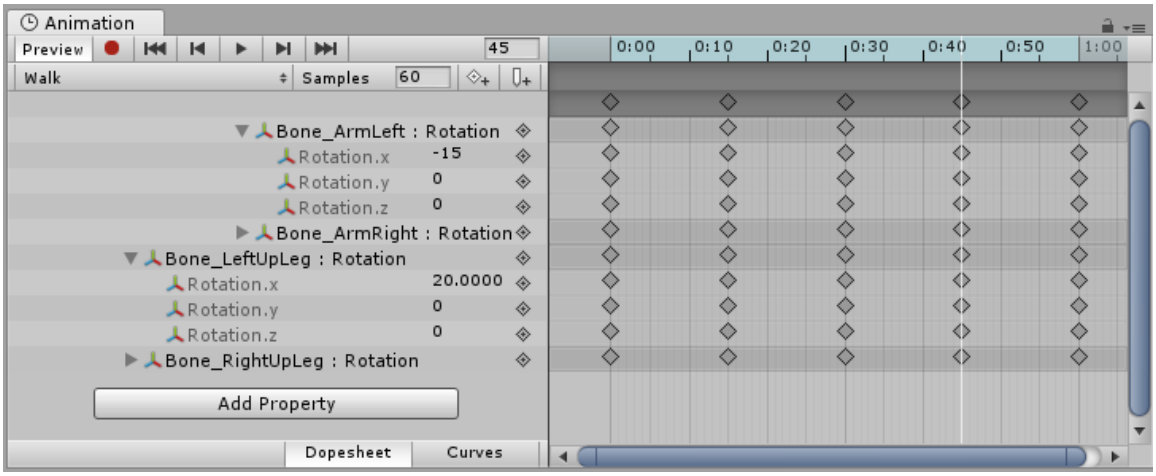
3D model postavy

Pro projekt byl potřeba 3D model postavy a velké množství různých animací. Najít si detailní 3D postavu od profesionálního grafika by nebyl problém, ale problém je, aby k dané postavě existovaly různorodé animace, které potřebujeme. Někoho by mohlo napadnout, vzít detailní postavičku (mesh + textury) a použít animace z jiných postav, ale to by nefungovalo. V Unity neexistuje pravidlo, jak by hierarchie kostí postavy měla vypadat. Měla by noha mít 3 kosti nebo 4? Z kolika kosti se má skládat páteř? Jaké mají být výchozí rotace kostí? Grafici (hlavně grafici co dělávají postavy do Unity) si dělají hierarchii kostí postavy podle svého. Pokud bych vzal různé animace určené původně pro různé postavy, animace by nefungovala, protože by buď vyžadovala kostí v postavě, která tam nejsou nebo naopak by existovali kosti, pro které nejsou v animaci klíčové snímky. Unity podporuje i *Retargeting* animací (jedna animace se použije na více postav o různých proporcích), ale *Retargeting* pořád vyžaduje, aby cílová postava měla posloupnost kostí stejnou jak postava, pro kterou byly animace vytvořeny. Z těchto důvodů jsem se rozhodl si vytvořit vlastní jednoduchou postavičku a vlastní animace v Unity v nástroji *Animation*.



Obrázek 3.7: model postavy

Postava nebyla vytvořena v externím programu, ale byla sestavena v editoru Unity z jednotlivých objektů, které reprezentují jednotlivé kosti postavy. Pod jednotlivé kosti jsem přiřadil další objekty obsahující komponenty jako jsou *BoxCollider* a *MeshRenderer* (kvá-



Obrázek 3.8: Nástroj *Animation* v Unity editoru pro tvorbu a úpravu animací. V nástroji na obrázku je otevřená jednodušší animace *Walk*. V levé částí jsou 4 kosti, které jsou animovány a na pravé straně jsou jednotlivé klíčové snímky animace. V tomto konkrétním příkladu je na obrázku dohromady 20 klíčových snímků (změna rotace 4 kostí v čase, kde každá kost má 5 klíčových snímků).

dry). Na obrázku 3.7 je zobrazena 3D postavu v náhledu při přehrávání animace. Obvykle ve hrách je tělo postavy tvořeno jedním modelem a nelze oddělit například ruku od zbytku těla. Oddělení by vyžadovalo existenci 2 modelů (ruky a zbytku těla). Jednou z výhod mé takto vytvořené postavy z částí je oddělitelnost libovolné části těla za běhu.

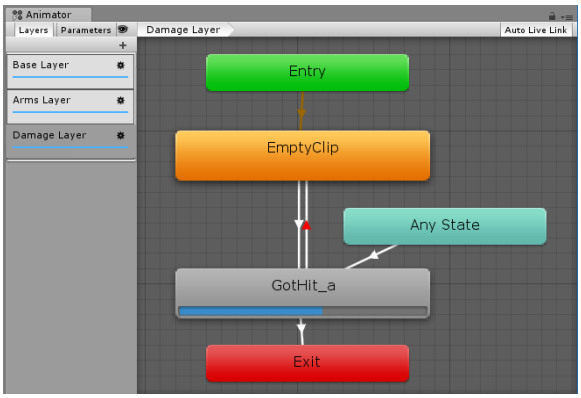
3.4 Animace

Animace [10] ve hrách jsou obvykle tvořeny z takzvaných animačních klipů (animation clip). Animačním klipem může být například chůze, skok či třeba úder rukou. Klip se může přehrát jednorázově nebo ve smyčce. Jedním z nejběžnějším použitím animačního klipu ve hrách je řízení pozice a rotace objektů. Jednou z výhod tvorby animace v rámci Unity je možnost animovat jakýkoliv veřejný atribut jakékoliv komponenty přidané na objektu. Obecně existují dva typy animací postav. Prvním typem jsou animace, které se přehrávají na místě (tzv. in place), kde animace nemění svoji pozici a nemá vliv na pohyb hráče. Pohyb hráče je v takovém případě založen například na fyzice engine. Druhým typem animací jsou pohyblivé animace, kde animace mění svoji pozici a tím je měněna i pozice hráče. V takovém případě pohyb hráče není založen na fyzice engine. Pohyblivé animace jsou běžné především u kvalitnějších (detailnějších) animací, které byly nahrány přes některý *MoCap* [7] systém.

Pro hru jsem si vytvořil vlastní animace přímo v Unity v okně (nástroji) *Animation*. Všechny animace jsou typu přehrávání na místě (in place). Pro svoji postavu jsem vytvořil celkem 28 jednodušších animací.

Pro přehrávání animací jsem použil vestavěnou komponentu *Animator* (neplést s *Animation*). Pro vytvoření stavového automatu animací s přechody (*ASM – Animation State Machine*) jsem použil nástroj *Animator*. Komponenta i stejnojmenný nástroj *Animator* jsou v Unity součástí systému *Mecanim* [10].

Na obrázku 3.9 je okno nástroje *Animator*, ve kterém jsem vytvořil víceúrovňoví stavoví automat animací. Na levé straně jsou 3 vrstvy (*Base Layer*, *Arms Layer* a *Damage Layer*). Na pravé straně je zobrazen stavový automat pro vybranou vrstvu *Damage Layer*. Každá vrstva ve stavovém automatu vždy obsahuje vstupní bod *Entry*, výstupní bod *Exit* a stav *Any State* pro možnost provedení přechodu do stavu libovolného ze stavu aktuálního.



Obrázek 3.9: Okno *Animator*; *ASM* – vrstva *Damage Layer*

Animator podporuje různé způsoby prolínání animací (blending). Prolínání animací je jeden z důležitých požadavků pro správné přehrávání dvou nebo více podobných animací na postavě. Prolínání animací nám například umožňuje, aby byla přehrávána animace chůze dopředu a zároveň chůze do strany a prováděna jiná akce jako je útok. Bez prolínání animací by v daný okamžik mohla být prováděna jen jedna akce, některá z chůzí nebo útok, nikoliv však všechny tři současně.

V Unity se mohou animace na animovaném objektu prolínat třemi způsoby:

- Animace mohou být členěny do vrstev, kde každá vrstva je přehrávána s jinou váhou.
- V rámci jedné vrstvy může dojít k přechodu (transition) mezi dvěma animacemi. V průběhu přehrávání jedné animace se začne přehrávat animace druhá. Dojde k plynulé interpolaci mezi dvěma animacemi.
- V rámci jedné vrstvy může být přehráváno více animací pomocí *Blend Trees*.

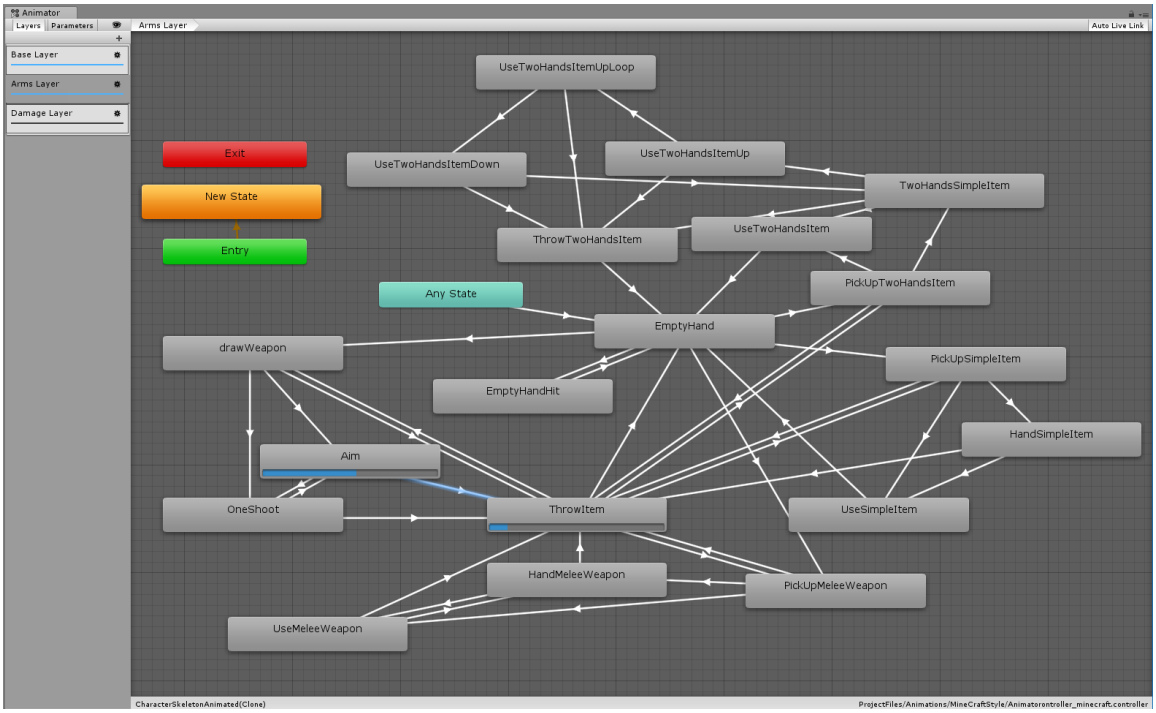
I když oba způsoby *Transitions* a *Blend Trees* mohou vypadat na první pohled podobně, jedná se o dva úplně odlišné způsoby prolínání animací a jejich použití je odlišné. Stavem v *ASM* není jen animační klip, ale taktéž i *Blend Trees*.

Prolínání vrstev

Jak už bylo zmíněno, stavy v *ASM* jsou členěny do vrstev. Vrstva s vyšším indexem se prolíná s vrstvou s indexem nižším podle nastavených vah. Má-li vrstva s vyšším indexem váhu rovnu jedné, tak úplně přepisuje společné atributy animací vrstvy nižší.

Přechody (Transitions)

Přechody se používají pro plynulý přechod z jednoho animačního stavu (*Animation State*) do jiného během určitém časového období. Mezi 2 animačními stavy vždy mohou existovat až 2 přechody (každý pro jeden směr). Každý přechod obsahuje podmínky, které musí být splněny, aby přechod započal. Přechody se používají například, když se dokončí přehrávání jedné animace a má plynule začít přehrávání animace jiné. Přechody jsou součástí každé vrstvy. Příklad s větším množstvím přechodů je na obrázku 3.10.



Obrázek 3.10: Okno *Animator*; *ASM* – vrstva *Arms Layer*. Obsahuje 19 animací (stavů) a přechody mezi nimi. Na obrázku je současným stavem *Aim* v přechodu do stavu *ThrowItem*. Vrstva slouží pro přehrávání animací používání předmětů na postavě.

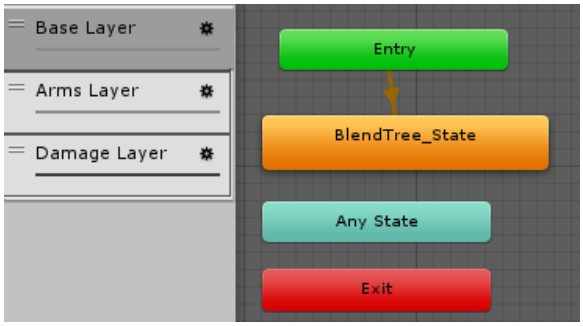
Blend Trees

Blend Trees se používají, když na jednom objektu v jedné vrstvě je potřeba přehrávat více animací současně. Například chůze dopředu a zároveň chůze do strany. To, jaká animace bude mít větší vliv je dán **parametrem prolnutí** v *Blend Tree*. Podle počtu parametrů prolnutí existují dva druhy *Blend Trees*:

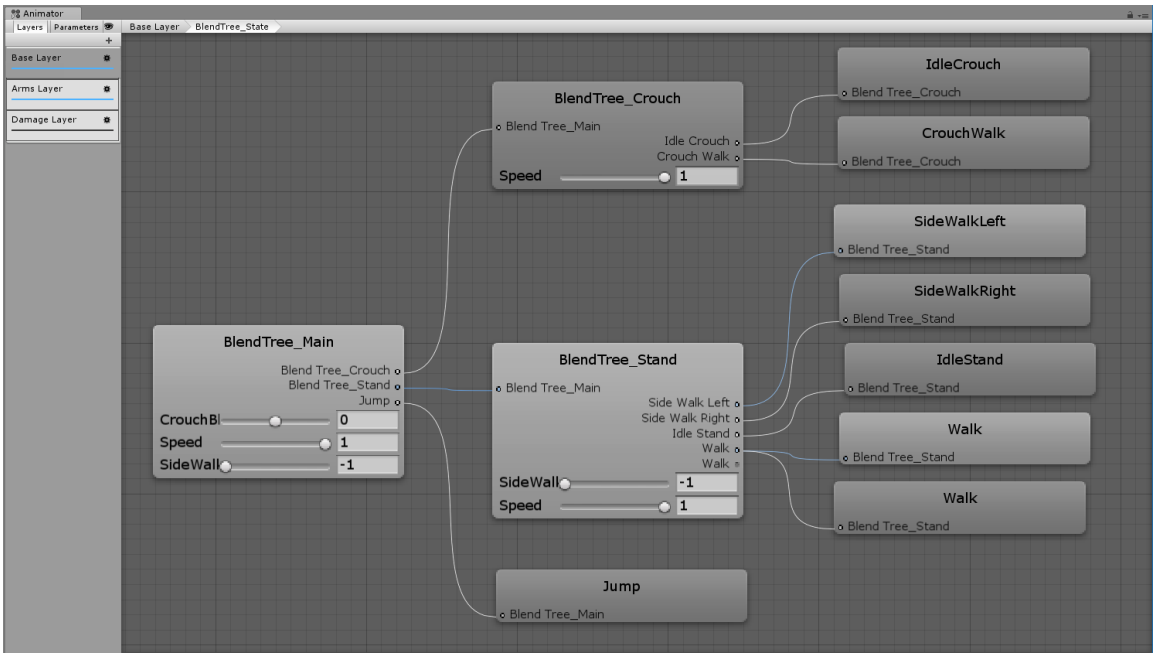
- **jednodimenzionální** – řízen jedním parametrem a umožňuje prolínat maximálně 2 animace současně.
- **dvoudimenzionální** – řízen dvěma parametry a teoreticky umožňuje prolínat neomezené množství animací současně.

Jak jednodimenzionální, tak dvoudimenzionální *Blend Tree* může řídit libovolný počet animací, ale jejich rozdíl je v maximálním počtu přehrávaných animací, které může daný *Blend Tree* v určitém okamžiku prolínat. Pro přehrávání dvou skupin animací v jedné vrstvě musel být vytvořen víceúrovňoví *Blend Tree*. Příklad víceúrovňového *Blend Tree* je na obrázku 3.12.

Na obrázku 3.11 je zobrazena základní první vrstva *ASM*. Při spuštění přehrávání animací, *Animator* automaticky přejde ze vstupního stavu do stavu *BlendTree_State*, ve kterém už zůstane. *BlendTree_State* je stav v sobě obsahující víceúrovňoví *Blend Tree* a animace (nikoliv další stavy).



Obrázek 3.11: Okno *Animator*; *ASM* – vrstva *Base Layer*. Vrstva slouží pro přehrávání animací reprezentující pohyb hráče (chůze, skok, skrčení, atd.).



Obrázek 3.12: Obsah stavu *BlendTree_State*. Jedná se o víceúrovňoví *Blend Tree*. *BlendTree_Main* a *BlendTree_Crouch* jsou jednodimenzionální. *BlendTree_Main* umožňuje prolínat *BlendTree_Crouch* s *BlendTree_Stand* nebo prolínat *BlendTree_Stand* s animací *Jump* (nikoliv *BlendTree_Crouch* s *Jump*). *BlendTree_Stand* je dvoudimenzionální *Blend Tree* a umožňuje prolínat až 3 animace současně. *BlendTree_Crouch* prolíná pouze animace *IdleCrouch* a *CrouchWalk*. V tomto konkrétním případě tento víceúrovňový *Blend Tree* dokáže prolínat až 5 animací současně. Například animace *IdleCrouch*, *CrouchWalk*, *SideWalkLeft*, *IdleStand* a *Walk*, když postava půjde dopředu, zároveň do boku a začne se u toho krčit.

3.5 Kamery

Hráč hraje hru z pohledu první osoby. Kamera je napevno umístěna do hierarchie postavy do výšky hlavy hráče. Kamera se pohybuje spolu s hráčem. Ve hře pro pohled hráče by stačila jedna kamera, ale může se stát, že předmět v ruce nebo část těla hráče bude zasahovat do jiného modelu ve scéně. Aby předmět nezasahoval do jiného modelu se nedá vždy zabránit.

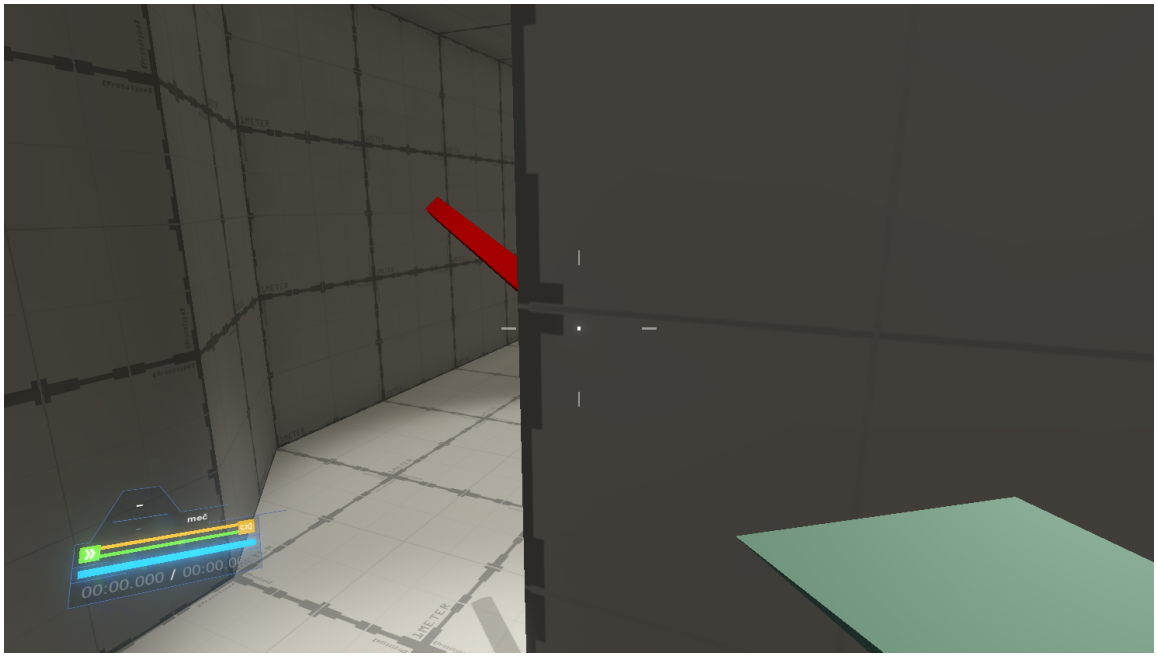
Řešením, aby z hráčova pohledu ruka a předmět v ruce nikdy nezasahoval do jiného předmětu ve scéně lze dosáhnout vytvořením dalších kamer, kde každá kamera bude vykreslovat jen určitě modely (vrstvy) a ve výsledném obraze pak předmět nebude zasahovat do jiného modelu.

Do hry jsem přidal další 2 kamery (celkem 3 pro hráče – všechny perspektivní) pro řízení pořadí vykreslení modelů. Původní kamera nejdřív do bufferu vykreslí většinu scény a pozadí. Vykreslí vše kromě hráče, předmětu v ruce a GUI (obrázek 3.14). Pak jedna z nově z přidaných kamer do bufferu vykreslí pouze předmět, který má hráč v ruce a hráčovu 3D postavu (i postava může zasahovat do jiného modelu) (obrázek 3.15). A na konec třetí kamera vykreslí do bufferu pouze GUI (obrázek 3.16). Můžeme vidět, že použitím více kamer je předmět vykreslován až po vykreslení scény a tím se předmět jeví, že do jiného modelu nezasahuje (obrázek 3.17). Tímto způsobem se nemůže stát, že by ruka nebo předmět v ruce byl vykreslován uvnitř jiného modelu. Stejný princip je použit i pro pohled z Ducha. Celkem je použito 6 kamer (3 v hráčovi a 3 v Duchovi).

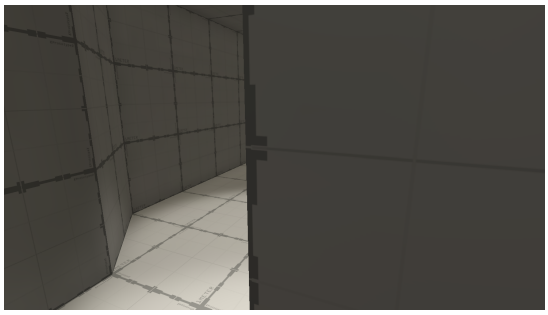
3.5.1 Post-processing

Unity obsahuje sadu vizuálních efektů aplikovatelných na buffer kamer. Vlastní efekty jsem v projektu nevytvářel, pouze použil existující. Použil jsem Bloom, Depth of Field, Chromatic Abberation, Vignette a Color Grading.

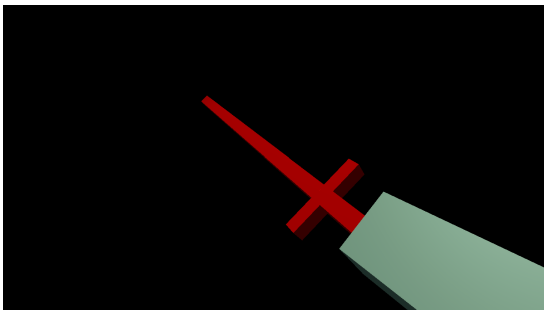
- Bloom je aplikován na buffer kamer po vykreslení GUI. Je dobře vidět na obrázku 3.16 (efekt svítivého GUI) a kolem laseru na obrázku 3.3.
- Color Grading a Chromatic Abberation používám při zpomalení času pro lepší efekt. Jsou aplikovány na buffer po vykreslení postavy (neovlivňují HUD). Color Grading umožňuje změnit hodnoty jednotlivých barevných kanálů RGB. Běžně se používá pro zvýšení kontrastu ve scénách. Používám ho pro snížení saturace obrazu. Naopak Chromatic Abberation obarví hrany objektů a mírně rozmaže obraz.
- Vignette vytváří černé okolí při režimu nahrávání hráčových událostí.



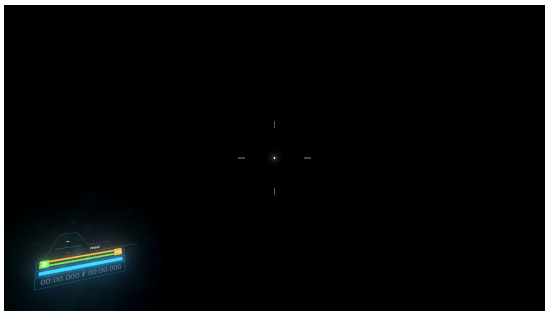
Obrázek 3.13: Snímek ze hry, za použití pouze jedné kamery, přibližně v půli přehrávání animace úderu se zbraní, kdy model zbraně a hráčovi ruky zasahují do rohu stěny.



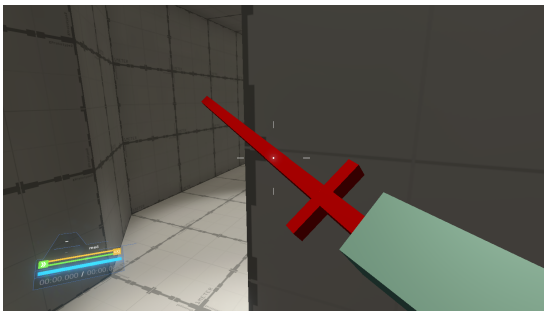
Obrázek 3.14: Kamera 1 (hlavní), vykreslení scény.



Obrázek 3.15: Kamera 2, vykreslení pouze hráče a předmětu v hráčovi ruce.



Obrázek 3.16: Kamera 3, vykreslení pouze GUI.

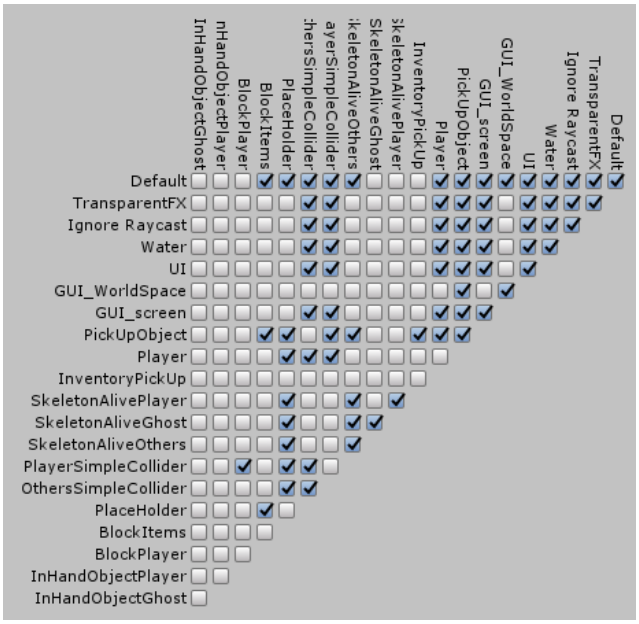


Obrázek 3.17: Výsledný obraz, který vidí hráč. Obraz je výsledkem sloučení obrazů jednotlivých kamer.

3.6 Vrstvy

Unity umožňuje přiřadit jednotlivé objekty na různé vrstvy [14]. Maximální počet vrstev je v Unity pevně dán. Vývojář má k dispozici až 32 vrstev, které může libovolně používat, přičemž ale jen 24 vrstev si může libovolně pojmenovat. Názvy prvních 8 vrstev nelze měnit (3 vrstvy dokonce nemají ani název, a tak je nelze přes editor používat ani upravovat). Možné využití vrstev jsme viděli v podkapitole s kamerami 3.5, kde každá kamera vykreslovala jen určité modely ve scéně, podle toho, na které byly vrstvě. Toto však ale není hlavní využití vrstev. Vrstvy se především používají k nastavení ve fyzice, které objekty spolu mají reagovat a které ne. Ve výchozím nastavení, spolu reagují všechny objekty a vývojář postupně volí, které objekty (vrstvy) spolu reagovat nemají.

Na obrázku 3.18 jsou zobrazeny všechny definované vrstvy v projektu a jestli objekty na daných vrstvách spolu reagují. Například *Capsule Collider* hráče i Ducha jsou na stejné vrstvě *PlayerSimpleCollider*. Vrstva *PlayerSimpleCollider* má nataveno, že objekty na této vrstvě spolu reagovat nemají. Proto, když se hráč vrátí do minulosti a Duch i hráč jsou na stejném místě, mohou být v sobě či sebou procházet bez jakéhokoliv omezení. Většina geometrie scény je na vrstvě *Default*. Vrstva *Default* je jedna z 8 předdefinovaných vrstev v editoru. Dále je nastaveno, že vrstva *Default* a vrstva *PlayerSimpleCollider* mají spolu reagovat. Kdyby tomu tak nebylo, hráč by ve hře propadl zemí či procházel stěnou.



Obrázek 3.18: Layer Collision Matrix z editoru Unity.

Bariéra zabraňující pronesení předmětu

Z hlediska návrhu scén jsem do hry přidal bariéry, které hráči znemožňují pronést předměty mezi určitými oblastmi. Hráč bariérou projít může, ale předmět přes bariéru nepronese. Hráč musí často vyřešit, jak do jiné oblasti předmět dostat. Pokud hráč vstoupí do bariéry s předmětem v ruce, inventář předmět odhodí. Taktéž předmět nelze bariérou prohodit. Bariéra se skládá ze dvou objektů. Jeden objekt na vrstvě *BlockItems*, aby hráč nemohl prohodit předmět bariérou. Druhý objekt (Collider) na vrstvě *IgnoreRaycast*, nastaven jako trigger, který při protnutí hráče přinutí hráčův inventář upustit předmět.

3.7 Grafické uživatelské rozhraní

Do hry jsem vytvořil HUD (head–up display) a menu. Unity podporuje dva způsoby (systémy), jak vytvářet grafické prvky:

- Systém GUI [13]: Grafické prvky se vytváří voláním statických metod v metodě *OnGUI* ve třídách *MonoBehaviour*. Metoda následně vrací svůj stav (např. tlačítko bylo zmáčknuto) (příklad ve výpisu 3.2). Jedná se o zastaralý způsob, kdy k překreslení všech grafických prvků dochází každým snímkem. Neobsahuje žádný grafický editor a práce s jednotlivými grafickými prvky je náročnější.

  Staré GUI vytvářelo elementy s absolutními pozicemi, neexistoval žádný automatický layout a ani náhled.

- Systém UI [16]: v roce 2015 byl do Unity přidán nový systém pro tvorbu grafického rozhraní, nazván pouze jako UI. Nový systém umožňuje vytvářet elementy přímo v editoru Unity. Elementy již nejsou vytvářeny voláním statických metod, ale přímo jako komponenty, které lze přiřadit na objekty ve scéně a lze na ně i odkazovat v kódu. Elementy jsou vykreslovány podobně jako ostatní polygony (modely) ve scéně. Elementům lze dokonce přiřadit materiál a shader.

Na obrázku 3.19 je z editoru zobrazena kamera a HUD, který kamera vykresluje (systém UI). Kdyby se do zorného pole před kameru přesunula např. bedna, tak by tato kamera bednu vykreslovala a bedna by překryla HUD. S prvky UI lze manipulovat stejně jak s ostatními objekty ve scéně. Nejen, že lze nastavit pozici elementů ve scéně, můžeme nastavit i jejich pozici a rotaci po všech třech osách. Na obrázcích 3.20 a 3.21 je dvakrát stejná část HUD, pravý je stejný jak leví jen s rozdílem, že pravý je natočen. Natočený HUD má díky perspektivní kameře lepší (pěkný) perspektivní efekt.

HUD se skládá pouze ze dvou typů elementů: Image a Text. HUD v levém dolním rohu displeje poskytuje hráči různé informace:

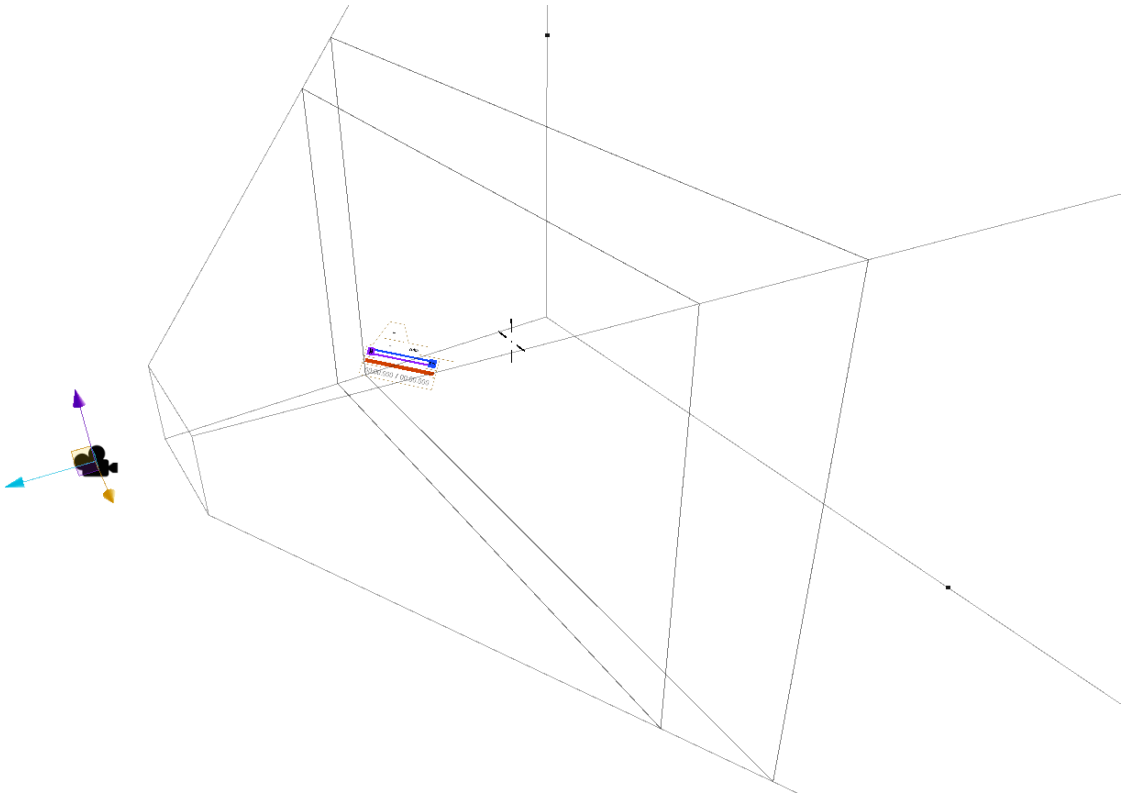
- Aktuální předmět v ruce: název a případně rozšiřující informaci. U střelné zbraně počet zbývajících nábojů v zásobníků ve zbrani.
- Stav svítilny (žlutý pruh).
- Běh (zelený pruh).
- Energie pro zpomalení času (modrý pruh).
- Doba nahrávání (současná délka nahrávky / maximální délka nahrávky).

Součástí HUD je zaměřovač (*Crosshair*) na středu obrazovky. Skládá se z tečky a čtyř proužků. Každý proužek je natočen směrem od středu. Zaměřovač zde plní více funkcí:

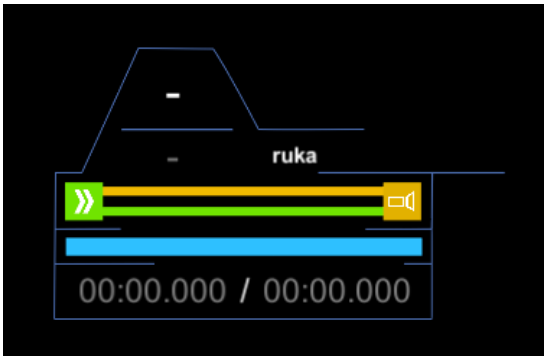
- Stejně jako v jiných hrách, pomáhá s mířením na cíl. Bez zaměřovače by hráč těžko odhadoval kam např. vystřelí.
- Zaměřovač taktéž vyjadřuje, zda je předmět v ruce používán. Když například vystřelíte nebo zvednete bednu, jednotlivé proužky zaměřovače se posunou od středu. Až se proužky vrátí zpět na střed, víte, že s předmětem můžete znovu dál manipulovat. Proužky posunuty od středu jsou vidět na obrázku 3.16.
- Tečka uprostřed mění barvu. Když je spuštěno nahrávání akcí, je červená. Při přehrávání akcí je zelená. Jinak bílá.

```
1 void OnGUI(){
2   // vykreslení tlačítka v levém horním rohu
3   if (GUI.Button(new Rect(10, 10, 100, 50), btnTexture)){
4       Debug.Log("Clicked the button with an image")
5   }
6 }
```

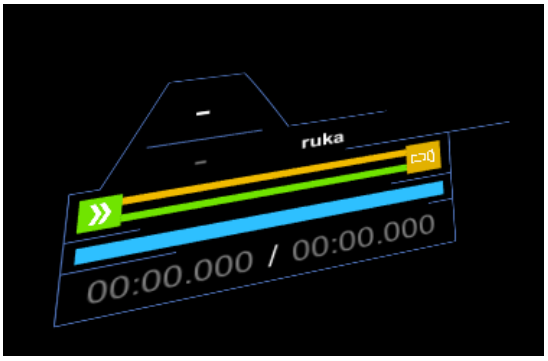
Výpis 3.2: Příklad tlačítka ve starém GUI



Obrázek 3.19: Kamera 3 – zobrazena v editoru. Obrázek má invertované barvy.



Obrázek 3.20: HUD v levém dolním rohu displeje.



Obrázek 3.21: HUD natočení o mínus 35 stupňů kolem vertikální osy a o 40 stupňů kolem horizontální osy.

Menu:

Ve všech scénách je dostupné menu. Menu se zobrazí, když hráč pozastaví hru. Menu se skládá z jednotlivých podmenu, mezi kterými se hráč přepíná. Menu hráči umožňuje:

- Spustit novou hru, načíst jakoukoliv úroveň, vrátit se do hlavního menu nebo vypnout hru.
- Nastavení hry: ztlumit zvuky ve hře a nastavit obraz (např. rozlišení, stíny a Anti–Aliasing)
- Rychlé načtení do záchytného bodu: umožňuje hráči se vrátit do některého z uložených stavů scény, vrátit se před bod nahrávání, znovu spustit nahrávání, znovu spustit přehrávání nebo se vrátit do uloženého stavu na konci přehrávání

Menu je responzivní. Přizpůsobuje se velikosti okna. Pokud by okno bylo až tak malé, že se jednotlivé řádky do okna nevejdou, zapne se vertikální posuvník.

3.8 Kolize

Unity podporuje detekci kolizí mezi objekty. Detekuje, když se dvě komponenty *Collider* začnou dotýkat, působí na sebe a přestanou se dotýkat. Jeden objekt se může skládat z více komponent *Collider*. Mezi dvěmi komponentami *Collider* může nastat libovolný počet kontaktů. Práci s kolizemi musí vývojář implementovat v metodách *OnCollisionEnter*, *OnCollisionStay* a *OnCollisionExit*.

```
1
2  public class CollisionCounter : MonoBehaviour {
3      private int numberOfContacts = 0;  // OK
4      private bool isInContact = false; // problém
5
6      void OnCollisionEnter(Collision collision) {
7        // nastala kolize s objektem
8          print("Object " + gameObject.name + " touched " + collision.gameObject.name);
9          isInContact = true;
10         numberOfContacts++;
11     }
12
13     void OnCollisionStay(Collision collision) {
14       // objekty jsou stále v kontaktu a dokonce na sebe působí
15         print("Number of contacts: " + collision.contacts.Length);
16         isInContact = true;
17     }
18
19     void OnCollisionExit(Collision collision) {
20         // skončila kolize
21         isInContact = false;
22         numberOfContacts--;
23     }
24 }
```

Výpis 3.3: Jednoduchý příklad detekce a počítání kolizí objektu s ostatními objekty ve scéně.

Když dojde ke kolizi dvou objektů, engine zavolá metodu *OnCollisionEnter* (může se zavolat víckrát v důsledku více kolizí). *OnCollisionExit* se zavolá, když jakákoliv z předešlých kolizí skončí (dva objekty se na určitém místě přestanou dotýkat). *OnCollisionStay* se volá každým snímkem fyziky, když dva objekty na sebe působí. Může nastat, že objekty jsou v kontaktu, ale *OnCollisionStay* se nevolá. *OnCollisionStay* se volá jen a pouze tehdy, když je alespoň jeden z objektů v pohybu (tzv. nespí) a zároveň se dotýkají.

V příkladu 3.3 se nemůžeme řídit hodnotou proměnné *isInContact*. Může dojít např. ke třem kolizím a jakmile by jedna kolize skončila, proměnná by nabyla špatné hodnoty. Pokud chceme vědět, zda objekt je v kolizi s nějakým jiným objektem, musíme kolize počítat. Proměnná *numberOfContacts* vyjadřuje, kolik celkem bodů dotyků existuje mezi objektem, kde je připojen komponenta *CollisionCounter* a ostatními objekty ve scéně. Tato proměnná nevyjadřuje, s kolika objekty je objekt v kontaktu. Jeden objekt může být v kontaktu např. s 5 objekty a mít celkem třeba 11 bodů kontaktu.

Pokročilá práce s kolizemi:

U některých objektů postačí znát, zda objekt je v kolizi, ale pokud chceme vědět kolik jich je a hlavně které to jsou, potřebujeme víc než jedno počitadlo. Například platforma pro modrou bednu potřebuje znát, zda je v kolizi s modrou bednou (s předmětem, který platformu aktivuje). Platforma je na vrstvě *Default*, modrá bedna, a i ostatní sebratelné předměty jsou na vrstvě *PickUpObject*. To znamená, že platforma reaguje s bednou, ale taktéž i se všemi ostatní sebratelnými předměty ve scéně. Z hlediska kolizí a vrstev nejsme schopni rozlišit, zda jsme na platformu položili bednu nebo třeba meč, ale nechceme, aby meč platformu aktivoval. To znamená, že musíme filtrovat objekty. Filtraci provádím na základě štítku (tag). Tag je jeden ze základních atributů (textový řetězec), které mají všechny objekt ve scéně.

```
Function OnCollisionEnter(collider cizího objektu) /* nastala kolize      */
    tag se neshoduje;
    if nebyl zadán žádný hledaný tag then /* nechceme filtrovat           */
        tag se shoduje;
    else
        for každý hledaný tag do
            if tag cizího objektu se shoduje s hledaným then
                tag se shoduje;
            end
        end
    end
    if tag se shoduje then
        předej správci informaci o nové kolizi;
    end
end
Function OnCollisionExit(collider cizího objektu) /* skončila kolize      */
    předej správci informace o ukončené kolizi;
end
```

Algorithm 1: Pseudokód komponenty *TriggerColliderChild*. Detekce a filtrace kolizí podle tag.

Filtraci podle štítků (tag) provádí komponenta *TriggerColliderChild*. Základní princip je popsán pseudokódem (viz. 1). Při vzniku nebo ukončení kolize předává informaci „správci“ – komponentě TriggerObjects. Pokud chceme znát, s jakými všemi objekty je náš objekt v kolizi, tak náš objekt musí mít komponentu *TriggerColliderChild* přidanou ve své hierarchii u každého objektu obsahující komponentu *Collider*.

```
Function addTransform(collider cizího objektu) /* někde v hierarchii cizího
 objektu nastala kolize s naším objektem                                  */
    najdi root cizího objektu;
    for každý zapamatovaný objekt do
        if root se shoduje se zapamatovaným objektem then
            collider nenalezen;
            for každý collider zapamatovaného objektu do
                if zapamatovaný collider je stejný jako cizí then
                    collider nalezen;
                    inkrementuj si počitadlo zapamatovaného collider;
                    break;
                end
            end
            if collider nenalezen then
                /* první kontakt s daným collider                       */
                zapamatuj si collider;
            end
            /* nejednalo se o první kolizi mezi objekty, ať se neprovede
               akce                                                     */
            return;
        end
    end
    zapamatuj si nový objekt (root);
    proveď akci;
    /* sem se napíše kód, který má provést, když se 2 objekty protnou
       */
    /* např. platformo otevři dveře                                     */
end
```

Algorithm 2: Pseudokód komponenty TriggedObjects pro uložení (zapamatování) informací o kolizích objektu s ostatními objekty. Základní principem je obyčejné počítání kontaktů s jinými objekty. Když vznikne kolize, počitadlo se zvýší. Odstranění (zapomenutí) informací o kolizích má stejný postup jak tento pseudokód, jen místo inkrementace je dekrementace. A místo zapamatuj by bylo zapomeň.

Komponenta *TriggedObjects* slouží pro rozeznání a uchovávání informací o kolizích s jinými objekty. Objekt, který má na sobě přidané komponenty *TriggedObjects* a *TriggerColliderChild*, vždy zná s kolika a se kterými objekty je v kontaktu. Dokáže od sebe jednotlivé objekty rozlišit (viz. pseudokód 2). Dané komponenty má na sobě přidané např. platforma pro modrou bednu, nikoliv však předmět modrá bedna. To znamená, že platforma ví, s kým je v kolizi, ale modrá bedna neví, s kým je v kolizi (vědět to ani nepotřebuje).

TriggedObjects taktéž funguje s používáním metod (událostí) *OnTriggerEnter* a *OnTriggerExit*. V Unity tyto dvě metody poskytují podobné informace, ale jsou jednoduší, protože mezi dvěma komponentami *Collider* nastaveny jako trigger může vždy dojít jen k jednomu

průniku na rozdíl od *OnCollisionXXX* metod, kde dochází k vícenásobným kolizím mezi dvěma komponentami *Collider*.

Problém při deaktivace či odstranění dynamických objektů:

Z pohledu vývojáře objekty ve scéně o sobě navzájem nevědí. Když nastane kolize mezi dvěma objekty, engine to objektům oznámí. To je v pořádku, ale problém nastává, když jeden objekt odkazuje na druhý, nebo si udržuje informace o počtu kolizí s ostatními objekty a my jeden z ostatních objektů vypneme nebo odstraníme. Při vypnutí nebo odstranění objektu se ostatní objekty nedozvědí, že jiný objekt byl vypnut či odstraněn. Nedojde k zavolání *OnCollisionExit* ani *OnTriggerExit*.

Ve výpisu 3.3 komponenta *CollisionCounter* měla jediný účel. Pamatovat si celkový počet kolizí s ostatními objekty. Představme si, že objekt s danou komponentou je v kolizi s jiným objektem a my ten jiný objekt odstraníme. Engine nezavolá *OnCollisionExit* a hodnota proměnné *numberOfContacts* se nesníží. Objekt si tedy pořád myslí, že je v kolizi, i když není! Problém taky vzniká, když bychom objekt neodstranili, ale jen vypnuli, popřípadě později znovu zapnuli. Taktéž se nebude detekovat opuštění kolize a při opětovném zapnutí objektu se v následující snímek fyziky opět bude detekovat kolize nová. To znamená, že proměnná *numberOfContacts* se už po druhé inkrementuje, ale ani jednou se nedekrementovala. To je problém, který si jako vývojáři musíme ohlídat. Možným řešením by bylo, kdyby objekt, který se má odstranit znal všechny objekty se kterýma je v kolizi, a oznámil jim, že si mají odstranit všechny odkazy na něj. Ale to by bylo příliš komplikované. Dalo by se použít komponentu *TriggedObjects* z pseudokódu 2, ale místo toho, aby danou komponentu měly na sobě jen objekty, které potřebují znát okolí, tak by tuto komponentu musely mít na sobě přidanou všechny objekty, které se mohou potencionálně odstranit nebo vypnout. Místo toho, aby komponenta *TriggedObjects* byla ve scéně třeba jen 15krát, byla by ve scéně třeba 200krát. I na objektech, které by ji nepotřebovaly, jen proto, aby si objekty odstranily odkazy mezi sebou. Kdyby engine volal metody *OnCollisionExit* a *OnTriggerExit* při vypnutí či odstranění objektů, problém by zmizel.

Řešení:
Vytvořil jsem novou komponentu *ObjectDeactivator*. Když má být odstraněn nebo vypnut objekt, tak se v ten moment neodstraní ani nevypne, ale dojde k požádání komponenty *ObjectDeactivator* o jeho vypnutí či odstranění. Komponenta přemístí objekt na jinou pozici mimo oblast ostatních objektů (třeba o 1 000 metrů po vertikální ose, kde není žádný objekt), počká dva snímky fyziky a až pak objekt vypne či odstraní. Kdyby se objekt přemístil a hned odstranil, pořád by se nezavolaly metody *OnCollisionExit* a *OnTriggerExit*. Ale když se počká dva snímky, engine mezitím správně dané dvě metody zavolá na všech objektech se kterýma byl objekt v kolizi a komponenty např. *TriggerColliderChild* a *CollisionCounter* si aktualizují ukončení kolize.

3.9 Nepřátelský voják

Herní mechanika zpomalení času je originální, ale její použití jen na zpomalování obyčejných objektů ve scéně nevyužívá naplno její herní potenciál. Proto jsem vytvořil i nepřátelského vojáka, jehož jediným cílem je zabit hráče (případně Ducha). Nepřátelský voják má podobný základ jako hráč. Může chodit po scéně a útočit na hráče.

Navigace AI:

Pro navigaci a pohyb vojáka se dá v Unity použít komponenta *NavMeshAgent* [15] a vygenerovat *NavMesh* [12] ve scéně. Komponenta slouží pro pohyb objektu ve scéně. Zadáte ji cíl a pokud existuje cesta v *NavMesh*, objekt se v průběhu času do cíle dostane. Ve scénách jsem *NavMesh* vygeneroval a použil, ale pro vojáka jsem *NavMeshAgent* nepoužil. Komponenta *NavMeshAgent* má řadu nevýhod:

- Kde není vygenerován *NavMesh*, tam se *NavMeshAgent* nedostane. Mezi dvěma nespojenými částmi v *NavMesh* musí existovat *Off-Mesh Link* [12].
- *NavMeshAgent* má společný cíl pro rotaci i pozici. Vždy otáčí objektem (vojákem) do následujícího bodu v cestě. Nemůže se otočit jinam, než kam jde.
- Jedním z parametrů komponenty je ukončovací vzdálenost pohybu. Když je vzdálenost mezi objektem s *NavMeshAgent* a cílem menší než ukončovací vzdálenost pohybu, objekt přestane chodit k cíli. To by možná znělo v pořádku, když je dostatečně blízko cíle, zastaví se. Bohužel to ale ovlivňuje i rotaci. Pokud je voják používající tuto komponentu úplně u hráče a voják pozastaví pohyb, pozastaví i otáčení. To znamená, že v mnoha případech se voják na hráče nedotočí. Možným řešením by bylo tento parametr nepoužívat, ale pak voják chodí příliš blízko k hráči a odsouvá ho.
- Nereaguje na vnější síly. Aktivní *NavMeshAgent* úplně přepisuje pozice a nelze kombinovat s komponentou *Rigidbody*.
- Dokáže chodit pouze dopředu. Žádná chůze do boku nebo poodstoupení od cíle, pokud je k cíli (k nepříteli) příliš blízko.

Proto v důsledku těchto nevýhod jsem vytvořil vlastní komponenty pro pohyb vojáka po scéně. Dvojce komponent *SoldierAI* a *NaviginationAI*. Komponenta SoldierAI volá akce např. útok, chůzi a otočení. Komponenta *NavigationAI* zná prostředí a ví např. zda vidí některý z cílů, zda na cíl lze zaútočit a zda se k cíli lze dostat.

Výhody komponent *SoldierAI* a *NaviginationAI*:

- Pokud mezi vojákem a cílem je přímý průchodný prostor, i když tam není vygenerován *NavMesh*, voják se do cíle dostane. Například, když je voják na vyvýšeném místě a musí pouze seskočit dolů. Taktéž lze pořád použít *Off-Mesh Links*.
- *NavigationAI* nepoužívá jen jeden typ cíle, ale má dva typy cílů – kam jít a kam být otočený. Když mezi vojákem a hráčem je překážka, zároveň voják vidí na hráče (překážka je nízká), tak voják překážku začne obcházet, ale zároveň na hráče zůstane otočený. Má-li v ruce střelnou zbraň, tak bude na hráče i střílet. *NavMeshAgent* by otáčel vojáka na následující bod v cestě, nikoliv na hráče.
- Nezáleží na vzdálenostech mezi hráčem a vojákem. Pokud voják hráče vidí, bude se na něj otáčet, i když je u hráče na maximální přiblížení.
- Reaguje na vnější síly fyziky. Voják má stejný základ jako hráč a používá komponentu *Rigidbody*. Kdyby do vojáka něco silně strčilo, vojáka to odhodí.
- Nechodí pouze dopředu, ale umí chodit i do boku a dozadu. Chůze do boku je vhodná, když voják musí obejít překážku, ale chce zůstat otočený na hráče. Chůze dozadu se

provádí, když je voják příliš blízko k hráči.
Příklad: Voják má v ruce meč. Voják si začne od hráče udržovat větší odstup, protože meč dokáže způsobit poškození na větší vzdálenost než prázdné ruce. Když si voják udržuje odstup od hráče je taktéž lepší pro herní zážitek. Voják, který je namáčknutý na hráči nepůsobí dobře.

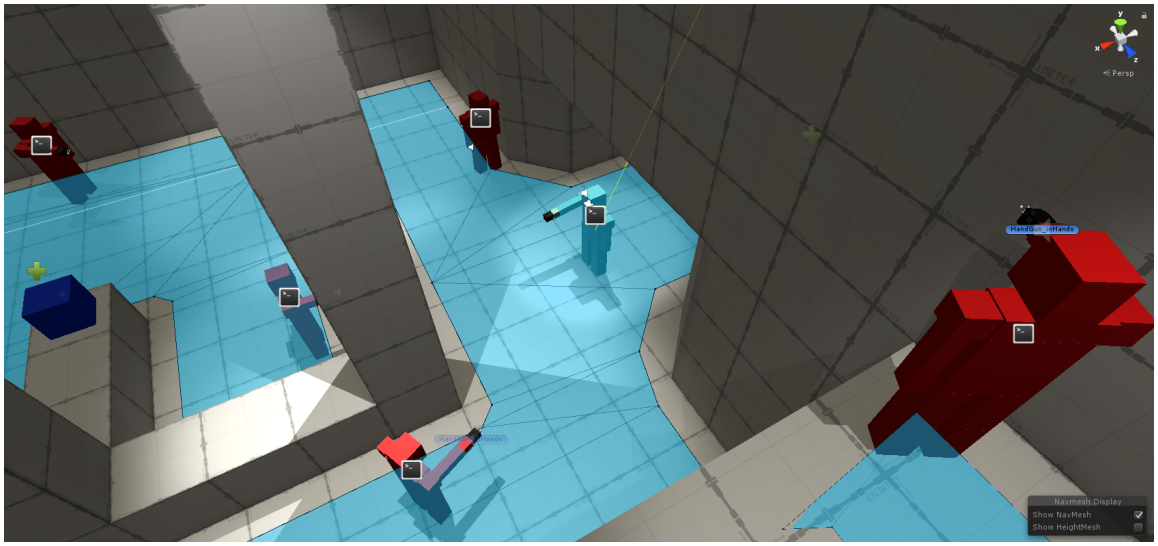
Pokud mezi vojákem a cílem (např. hráčem) existuje cesta v *NavMesh*, voják se vždy může do cíle dostat. Cesta je posloupnost bodů. Prvním bodem je vždy voják, posledním bodem je cíl. Ostatní body jsou rohy v *NavMesh*. Voják jde vždy do tzv. následujícího bodu. Následujícím bodem je obvykle druhý bod v cestě. V případě malých vzdáleností mezi body v cestě je následující bod ten, který je první vzdálenější od vojáka více než 20 cm. Pokud vidí nepřítele, je rovnou otočen na nepřítele, jinak je otočen do následujícího bodu. Jednoduchý příklad práce s *NavMesh* je ve výpisu 3.4, příklad vygenerovaného *NavMesh* na obr. 3.22.

```
1
2     // cesta navigačního systému Unity mezi dvěma body, zatím prázdná
3     NavMeshPath path = new NavMeshPath();
4
5     // získání cesty mezi dvěma body (mezi vojákem a cílem) - vypočte engine
6     NavMesh.CalculatePath(transform.position, targetPoint.position, NavMesh.AllAreas,
          path);
7
8
9     // následující bod v cestě
10    Vector3 nextPoint = Vector3.zero;
11
12    if(path.corners.Length > 2){
13      // cesta není přímá, obsahuje rohy, následujícím bodem může být první roh v cestě
14      nextPoint = path.corners[1];
15    }else if (path.corners.Length == 2) {
16      // existuje přímá cesta
17      nextPoint = targetPoint.position;
18    }else{
19      // cesta k cíli neexistuje
20      nextPoint = transform.position;
21    }
```

Výpis 3.4: Jednoduchý příklad s NavMeshPath. Získání následujícího bodu v cestě k cíli.

Životy postav

Každá postava má určité množství životů. Tělo postavy se skládá z částí. Podle toho do jaké části postavy je způsobeno poškození, ubere se podle typu části těla množství životů. Střední část těla přijímá 100 % z poškození. Hlava přijímá 125 % z poškození. Ruce a nohy přijímají 50 % a 40 % z poškození. To znamená že postava dřív umře, když bude dostávat zásahy do hlavy oproti zásahům do nohy. Každá zbraň ubírá jiné množství životů. Voják zabije hráče na jeden zásah jakoukoliv zbraní. Hráč zabije vojáka na jeden či více zásahů podle typu použité zbraně a místa zásahu.



Obrázek 3.22: Snímek z editoru. Na obrázku je vidět vygenerovaný *NavMesh* (modrý povrch podlahy) a pět nepřátel (červené postavy) kolem hráče. Kdyby hráč střelil do ruky vojáka, který je na obrázku nejvíce vpravo, voják by upustil zbraň, seskočil za hráčem a hráče by dál pronásledoval.

3.10 Zpomalení času

Bez zpomalení času rychlost hry odpovídá rychlosti reálného času. Jedna sekunda v reálném světě odpovídá jedné sekundě ve hře. Hráč má schopnost zpomalit rychlost hry přibližně 142krát. Pak jedna sekunda reálného času odpovídá přibližně 0,007 sekund ve hře. Na nejvyšší úrovní, zpomalení času řídí moje komponenta *TimeManager*. Její hlavním účelem je počítat novou rychlost hry v průběhu času.

Možný způsob řešení – třída *Time*

Hru v Unity si můžeme představit jako simulaci, která běží určitou rychlostí. Rychlost hry můžeme měnit nastavením statických proměnných *timeScale* a *fixedDeltaTime* ve třídě *Time*. Změnou proměnné *timeScale* se mění časový krok. Hra bude mít pořád přibližně stejný počet vykreslených snímků za sekundu. Efekty, animace a ostatní komponenty budou pracovat s novým časovým krokem správně, ale změní se počet snímků fyziky za sekundu. Aby počet snímků fyziky za sekundu zůstal stejný, proměnná *fixedDeltaTime* musí nabývat hodnotu doby reálného času mezi dvěma snímky fyziky krát simulační rychlost. Konkrétní příklad je ve výpisu 3.5.

```
1 // zpomalit rychlost hry např. 10krát
2 Time.timeScale = 0.1f;
3
4 // čas mezi dvěma snímky fyziky je ve výchozím nastavení 20 ms
5 Time.fixedDeltaTime = 0.1f * 0.02f;
```

Výpis 3.5: Změna rychlosti času ve hře. Konkrétní příklad, jak zpomalit čas 10krát.

Pokud bychom proměnnou *fixedDeltaTime* neměnili úměrně k rychlosti hry a rychlost hry by byla například 1000krát menší, snímek fyziky by se provedl jednou za 20 sekund

místo každý 20 milisekund. To by znamenalo, že by všechny dynamické objekty, co mají komponentu *Rigidbody* (např. hráč) by se posunuly jednou za 20 sekund místo 50krát za sekundu.

Použitý způsob řešení – přepočet hodnot v komponentách

Změna těchto dvou proměnných potencionálně stačí pro schopnost zpomalení času. V projektu jsem ji měl jednu dobu i takto implementovanou pro odzkoušení, ale ve finále jsem se rozhodl pro jiný způsob implementace schopnosti zpomalení času. Řídit rychlost hry přes proměnné ve třídě *Time* je relativně jednoduché, kód krátký, ale má to základní obrovskou nevýhodu. Změna kroku ovlivní úplně všechny objekty ve scéně. Takže nejen efekty a pohyb ostatních dynamických objektů ve scéně, ale i pohyb hráče. Stejně jak okolí by bylo zpomaleno, tak i hráč by byl zpomalen, a to se neslučuje s herní mechanikou, kterou jsem navrhl. Hráč nesmí být zpomalen, když okolí je zpomaleno. To znamená, že statické proměnné ve třídě *Time* neměním a efekty zpomalení času jsem implementoval zvlášť pro všechny typy komponent. Různé komponenty implementují metody *initTimeScale* a *onTimeScaleChanged* z rozhraní *ITimeScaleChange*. Obsahem metod je přepočet aktuálních hodnot komponent podle rychlosti hry.

```
1
2  [RequireComponent(typeof(TimeObject))]
3  public class RigidbodyTime : MonoBehaviour, ITimeScaleChange {
4
5    // Unity komponenta pro simulaci fyziky
6      public Rigidbody rig;
7
8      void Awake() {
9          initRig(); // inicializace
10     }
11
12     private void initRig() {
13         if (rig == null) {
14             rig = gameObject.GetComponent<Rigidbody>();
15         }
16         /* vypnout automatickou gravitaci, gravitace totiž musí být úměrná k rychlosti
               hry */
17         rig.useGravity = false;
18     }
19
20     void FixedUpdate() { //engine volá každým snímkem fyziky
21         // aplikovat gravitaci pouze na daném objektu, úměrně k rychlosti hry
22           float currTimeScale = TimeManager.getTimeScale();
23           float dt = Time.fixedDeltaTime * currTimeScale * currTimeScale;
24           rig.velocity += Physics.gravity * dt;
25     }
26
27     public void initTimeScale() {
28         initRig();
29         changeSimulationSpeed(TimeManager.getTimeScale());
30     }
31
32     public void onTimeScaleChanged() {
33         changeSimulationSpeed(TimeManager.getTimeScaleDiffRatio());
34   }
35
36     private void changeSimulationSpeed(float newTimeScale) {
```

```
37
38          float timeScale = TimeManager.getTimeScale();
39
40          if (timeScale < 0.9f) {
41           /* minimalizace nežádoucí chyby engine (nastavením omezení maximální rychlosti,
                 kterou může dát engine objektu při kolizi) */
42              rig.maxDepenetrationVelocity = 0.2f * TimeManager.getTimeScale();
43          } else {
44            // nastaveni původní hodnoty
45              rig.maxDepenetrationVelocity = 1e+32f;
46          }
47
48      // mass je hmotnost objektu
49      /* hmotnost nemá vliv na aktuální rychlost objektu, ale měním hmotnost z důvodu, aby
             hráč neovlivňoval tolik předměty, když do nich narazí ve zpomaleném čase */
50          rig.mass /= (newTimeScale * newTimeScale);
51
52        /* přepočítaní hodnot komponenty, čím pomalejší čas, tím se objekt musí pohybovat
             pomaleji */
53          rig.velocity *= newTimeScale;
54          rig.angularVelocity *= newTimeScale;
55
56      // i odpor musí být nižší, když se zpomalí hra
57          rig.drag *= newTimeScale;
58          rig.angularDrag *= newTimeScale;
59      }
60  }
```

Výpis 3.6: Ovládání rychlosti pohybu dynamického objektu úměrně ke zpomalení času.

Výpis 3.6 je příklad mé existující fungující komponenty pro zpomalení objektu. Je navržena a vytvořena tak, že ji stačí přiřadit na objekt, kde je *Rigidbody* a objekt bude měnit rychlost podle rychlosti času. U objektů využívající fyziku engine Unity se musí se změnou rychlosti hry měnit hodnoty komponenty *Rigidbody* a nepoužívat automatickou globální gravitaci. Gravitaci musíme aplikovat zvlášť každým snímkem fyziky přidáváním rychlosti objektu.

Problém: Ve fyzice v Unity existuje určitý problém při zpomalení času [21]. Když je manipulováno s časem, objekty s *Rigidbody* mohou dostat vysokou novou rychlost. Místo rychlosti např. 1 cm/s budou mít rychlost 1 m/s. Nezáleží ani zda je zpomalení času implementováno přes statické proměnné v *Time* nebo mým způsob přepočítáním hodnot na každé komponentě zvlášť. Oba způsoby trpí stejným problémem (testoval jsem oba způsoby).

Projevení problému: Objekt dostane výrazně vysokou novou rychlost, pokud je zpomalen čas, objekt je v kolizi s jiným objektem a někdy později se čas opět zrychlí. Engine nepřepočítává sílu, kterou má dát objektům v kolizi, když se manipuluje s rychlostí hry. Ať je rychlost hry jedna nebo stokrát menší, nová síla je vždy stejná.

Jádrem problému je tíhové zrychlení působící na objekty. Ať už objekt padá k zemi nebo leží na zemi, každým snímkem fyziky objekt dostane novou rychlost a posune se. Když se dva objekty protínají, engine objekty posune od sebe a přidá jim novou rychlost ve směru od kolize. Proto ve hrách můžeme často vidět, když je více dynamických objektů na sobě (třeba bedny), že se někdy malinko třepou nebo poskakují. Engine je posouvá do sebe (ve směru gravitace), ale jelikož leží na sobě, dojde k protnutí a engine je zas od sebe posune (a tak pořád dokola). Když se objekt pohybuje vyšší rychlostí a srazí se s jiným objektem,

výsledkem bývá viditelné odražení objektů od sebe. Když ale objekt leží na zemi, nám připadá, že se objekt nehýbe. I objekty na zemi se ve skutečnosti hýbou, ale vždy když se posunou směrem do země, engine je posune směrem od země. Obvykle je ve výsledku objekt posunut do skoro identického místa kde byl, novou sílu utlumí gravitace v dalším snímku a pozorovateli to připadá, že se objekt neposouvá.

Na obrázku 3.23 je zobrazen zjednodušený princip fyziky engine při posunu objektu ležícího na jiném objektu (bez manipulace času nevzniká problém). Problém fyziky engine se projeví, když se manipuluje s rychlostí hry a nastane kolize, viz obr. 3.24.

V engine Unity lze v *Physics Manager* [14] nastavit, za jakých podmínek mají být objekty od sebe odráženy, když se protínají. Například, jak moc hluboko mohou být objekty v sobě, než je engine Unity posune od sebe. Avšak změna těchhle podmínek nepomůže. Naopak objekty mohou začít sebou procházet, protože engine je přestane od sebe rozrážet. Pokud vynutíte maximální sílu odrazů na 0, když dva objekty jsou v sobě, pořád se budou rozrážet. *Physics Manager* taktéž ovlivňuje nastavení fyziky celé hry, nikoliv určitých objektů. Takže i kdyby ve *Physics Manager* bylo řešení, pro moji herní mechaniku zpomalení času by to použít nešlo.

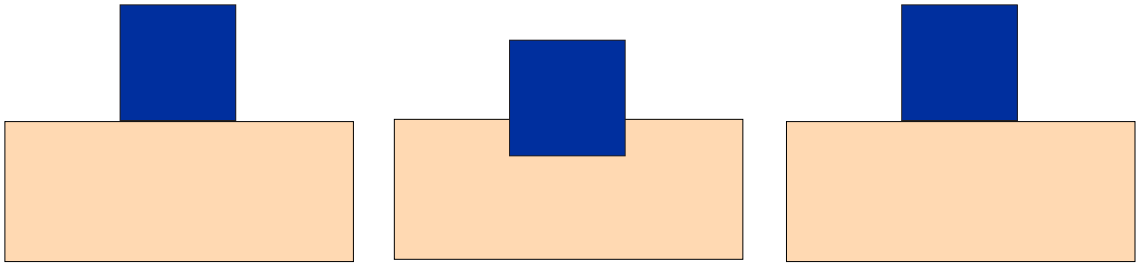
Problém se dá řešit dvěma způsoby:

- Interpolací rychlosti času v průběhu času. Pokud hra není zpomalena v nulovém čase, ale zpomaluje se postupně v průběhu času. Ukázka je ve výpisu 3.7.
- Úpravou parametru *maxDepenetrationVelocity* komponenty *Rigidbody*. Tento parametr vyjadřuje, jaká může být maximální rychlost, kterou engine přidá objektu, když je v kolizi s jiným objektem. Výchozí hodnota parametru je 10^{32}. To je extrémně vysoká hodnota, takže engine prakticky není omezen, jakou rychlost má aplikovat na objekt při kolizi. Snížením této hodnoty na hodnotu závislou na rychlosti hry se sníží nežádoucí efekt rozrážení objektů při manipulaci s časem. Příklad změny tohoto parametru je ve výpisu 3.6.

Těmito dvěma způsoby se problému úplně nezbavíme, ale výrazně ho minimalizujeme. Uživateli, kterému o daném problému nepovíme, si ho pak ve hře obvykle ani nevšimne.

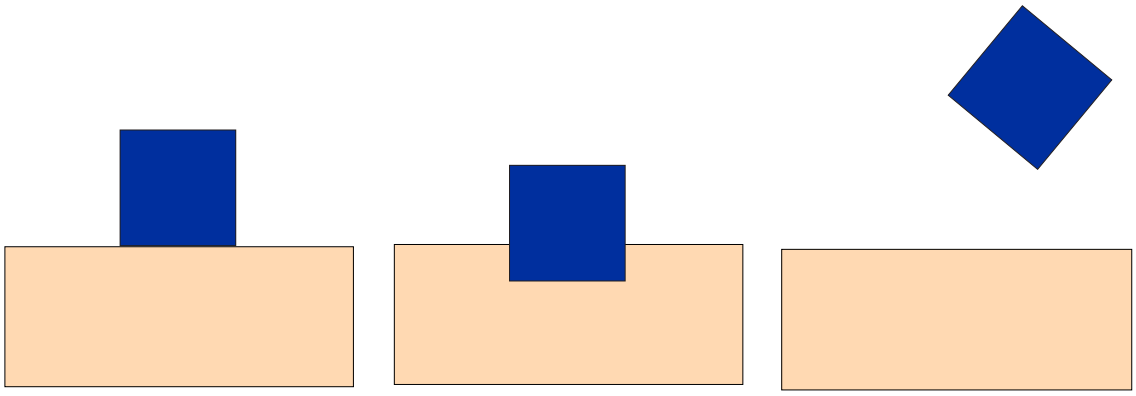
```
1    float prevTimeScale = currTimeScale;
2    // změna rychlosti hry
3    currTimeScale = Mathf.Lerp(currTimeScale, targetTimeScale, smoothTimeScale * 50f *
         Time.fixedDeltaTime);
4
5    // výpočet hodnoty, kterou se později vynásobí všechny hodnoty měnící rychlost
6    // např. rig.velocity *= timeScaleDiffRatio;
7    timeScaleDiffRatio = currTimeScale / prevTimeScale;
8
9    // onTimeScaleChanged je typu delegate event, obsahuje odkazy na všechny metody na
10   //všech objektech, které se mají provést v důsledku změny rychlosti hry (např metoda
         onTimeScaleChanged v RigidbodyTime)*/
11   if (onTimeScaleChanged != null) {
12     onTimeScaleChanged();
13   }
14   // při zpomalení času taktéž zpomaluji přehrávané zvuky
15   if (globalAudioManager) {
16     globalAudioManager.setGlobalPitchNotLinear(currTimeScale);
17   }
```

Výpis 3.7: Blok kódu ze třídy *TimeManager* prováděného každý snímek fyziky. Výpočet nové rychlosti času v průběhu času a zavolání události v důsledku změny rychlosti hry.



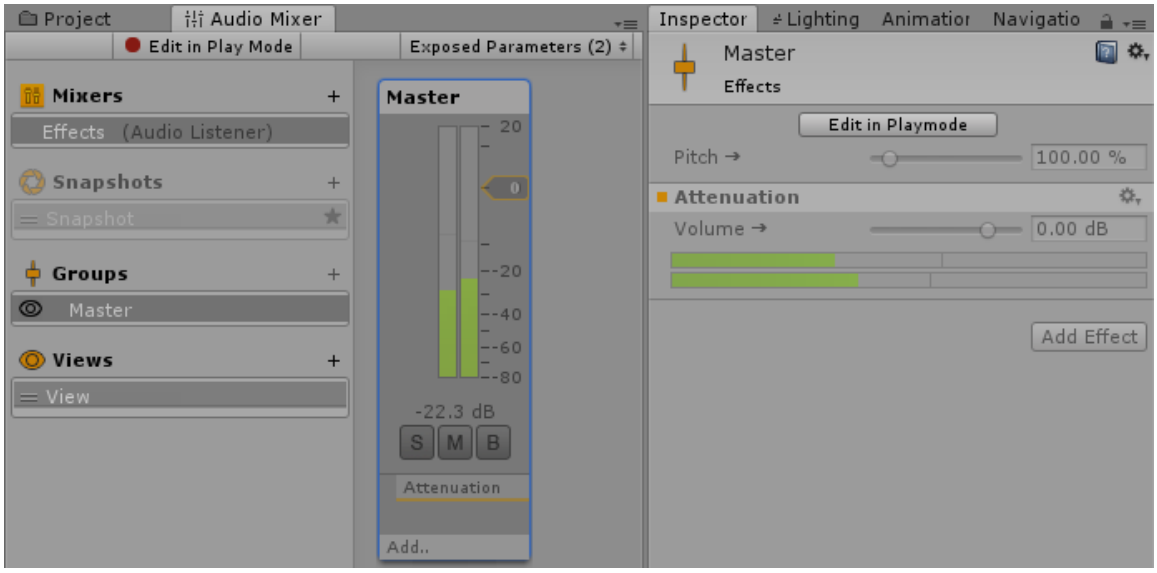
(a) bedna nad zemí (b) bedna posunuta do země (c) bedna opět na zemi

Obrázek 3.23: Chování bedny ležící na zemi, když není manipulováno s časem – nevzniká problém. Na zemi leží modrá bedna (a). V rámci jednoho snímku fyziky je na bednu aplikována gravitace, bedna se posune do země (b), engine bednu ze země posune zpět nahoru(c) a přidá novou rychlost. Bedna se vrátí přibližně na stejné místo, kde byla než se snímek fyziky provedl. Následující snímek fyziky by byla opět aplikována gravitace, bedna by opět protnula zem a opět by byla posunuta nahoru.



(a) bedna nad zemí (b) bedna posunuta do země (c) bedna vystřelená od země

Obrázek 3.24: Chování bedny ležící na zemi, když je manipulováno s časem – vzniká problém. Mějme stejný základ jak na minulém obrázku 3.23. Na zemi leží modrá bedna (a). V rámci jednoho snímku fyziky je na bednu aplikována gravitace, posune se do země (b), engine bednu ze země posune zpět nahoru(a) a přidá novou rychlost. Bedna se opět vrátí přibližně na stejné místo kde byla, než se snímek fyziky provedl. Pokud ale před tímto snímkem byl zpomalen čas a po tomto snímku se čas zrychlí, vznikne problém. Pokud bedna dostala novou rychlost např. 0,01 m/s, když byl čas zpomalen, až čas bude rychlejší, bude mít rychlost 1,4 m/s. Bedna v dalším snímku fyziky nebude opět posunuta do země, ale vystřelena směrem od země (c).



Obrázek 3.25: Audio Mixer z editoru Unity. Ovládá hlasitost a rychlost přehrávání všech zvuků ve hře.

3.11 Zvuky

Hra bez zvuků by působila neúplně. Zvuky dokáži vylepšit atmosféru a dojem ze hry. Ozvučil jsem například výstřel zbraně, úder předmětem, úder rukou, generátor laseru, dopad laseru, dopady a odrazy kulky. Pro globální ovládání hlasitosti a rychlosti přehrávání zvuků jsem použil *Audio Mixer* [11] z Unity. *Audio Mixer* řídím svojí komponentou *GlobalAudioManager*. Jednotlivé zvuky jsou přehrávány Unity komponentou *Audio Source*.

```
1  [SerializeField] private AudioMixer mainAudioMixer;
2  [SerializeField] private AnimationCurve pitchCurve = new AnimationCurve(new Keyframe
3  (0f, 0.3f), new Keyframe(1f, 1f));
4
5  // voláno při změně rychlosti hry
6  public void setGlobalPitchNotLinear(float t){
7    setGlobalPitch(pitchCurve.Evaluate (t));
8  }
9
10 // nastaví novou rychlost přehrávání zvuků
11 public void setGlobalPitch(float newPitch) {
12   globalAudioPitch = newPitch;
13   mainAudioMixer.SetFloat("MainPitch", globalAudioPitch);
14 }
```

Výpis 3.8: Část třídy GlobalAudioManager; metody pro změnu rychlosti přehrávání zvuků.

Když hráč pozastaví hru, zavolá se metoda *setGlobalPitch* pro pozastavení přehrávání zvuků. Při změně rychlosti hry se volá metoda *setGlobalPitchNotLinear*. Existují dvě různé metody pro změnu rychlosti přehrávání zvuků, protože je nevhodné, když hra je 142krát zpomalena, aby i zvuk byl zpomalen 142krát. Jedná se o tak velké zpomalení, že zvuk se přehrává po jednotlivých tónech. To pro člověka může být nepříjemné a ani nerozezná zvuky, které se přehrávají. Proto, když hra je zpomalena například 142krát (časový krok

je 0,007 s), rychlost přehrávání zvuků není 0,007krát, ale 0,3krát. Při zpomaleném čase se tím pádem zvuk přehraje rychleji, než má, ale hráč si toho ani nepovšimne.

3.12 Cestování zpět v čase

Základní princip a jednoduchý příklad byl popsán v podkapitole s herními mechanikami 2.2.

Z pohledu hráče můžeme rozdělovat hru do třech fází:

- Fáze nahrávání,
- Fáze přehrávání,
- Fáze nečinnosti.

Cestování zpět v čase umožňuje procházet všemi prostory, které jsou potencionálně uzavřeny. Při nahrávání jsou všechny dveře průchodné a hráč je nesmrtelný. Hráč může cestovat zpět v čase jen v určitých oblastech. Je to omezení z důvodu návrhu úrovní. V režimu nahrávání by hráč mohl projít celou scénu, protože všechny dveře jsou pro něj potencionálně otevřeny. To by vedlo na příliš velké bloudění uživatele po scéně a nošení předmětů z budoucnosti, které třeba ani nepotřebuje. Zda dveře jsou skutečně uzavřeny nebo se je hráči podaří otevřít a Duch bude moct projít se rozhodne až podle akcí hráče ve fázi přehrávání.

3.12.1 Fáze nahrávání

Hráč v libovolný moment může vytvořit bod v čase, do kterého se později bude chtít vrátit. Vytvořením bodu v čase se uloží stav scény a započne nahrávání hráčových akcí (událostí). Uložení stavu scény neznamená zkopírování celé scény, ale ukládají se jen a pouze určité proměnné o kterých se ví, že během hrání mohou měnit svoji hodnotu a potřebují být uloženy. Ne všechny proměnné, které mění hodnotu musíme ukládat. Ukládání se neprovádí do souboru. Zapamatované hodny existují jen v paměti po dobu běhu programu. V průběhu nahrávání se zaznamenávají všechny akce, které hráč chce provést. Akce nemusí být ani provedena, zaznamená se, že vůbec danou akci hráč chtěl provést. Ve fázi nahrávání akce ještě totiž nemusí být ani proveditelná, ale ve fázi přehrávání až Duch akce bude opakovat a vznikne časový paradox, akce může být proveditelná.

Fáze nahrávání a přehrávání spravuje komponenta *Recorder*. Komponenta obsahuje odkaz na první uloženou událost. Události tvoří lineární seznam. Každá událost má zaznamenána čas vytvoření. V pořadí, v jakém byly události uloženy jsou i přehrávány. Nestačí uložit, jen jaký typ události hráč provedl, každá událost potřebuje různé informace pro její opětovné přesné zopakování. Každá událost potřebuje znát, na které postavě se událost provede (v našem případě vždy na Duchovi).

Typy událostí:

- *ObjectControllerAxisEvent*: otáčení objektů (např. zrcadlo na stojanu). Nestačí uložit, že se má akce provést. Čas mezi snímky je vždy jiný. Musí se uložit celková síla otočení po horizontální ose a po vertikální ose. Jinak až by Duch akci opakoval a čas mezi snímky by byl odlišný, akce by měla jiný výsledek.
- *InventoryEvent*: akce inventáře. Dohromady existuje 7 typů akcí inventáře.
- *SpecialMovementEvent*: skok a dřepnutí (stoupnutí).

- *FlashlightEvent*: zapnutí nebo vypnutí baterky.
- *SimpleMovementEvent*: interpolace pozice Ducha v průběhu času. Interpolovaná pozice se počítá z hodnot záznamu současného a záznamu následujícího. Bez interpolace by se pohyb Ducha zdál neplynulý. Výrazně by to bylo vidět při zpomalení času.

Hráčův pohyb je založen na fyzice engine Unity. Duch je pouze přemisťován na místa, kde byl hráč v průběhu času. Je potřeba, když má být aplikována událost např. inventáře, aby Duch byl stejném místě, kde byl hráč, když událost byla zaznamenána. I malý rozdíl pozic a úhlů může vést k nesplnění podmínek pro provedení události, která měla nastat v určitém čase (např. sebrání předmětu). Nelze opakovat aplikování sil na Ducha stejným způsobem jak na hráče. Při přehrávání engine bude aplikovat síly neidenticky, popřípadě by mohla nastat s Duchem kolize, která předtím nenastala s hráčem. Takže by Duch nabýval různých rychlostí než hráč při nahrávání. Duch by se mohl např. opozdit nebo bychom ho mohli shodit třeba z vyvýšeného místa. Tím by Duch byl jinde, než má být a nesplnily by se podmínky pro úspěšné provedení událostí. Pokud by Duch spadl, kde hráč nespadl, v ten moment by Duch už nebyl vůbec schopen procházet stejnými místy, kterými šel hráč. Příklady vytvoření některých událostí jsou ve výpisu 3.9.

```
1   float deltaTime = TimeManager.getDeltaTimeSlowMotion();
2
3     /* V engine detekce zmáčknutí kláves nefunguje v podobě událostí, ale dotazováním.
          Hráč se dotazuje na stav kláves. */
4   float horizontalAxis = PlayerInputs.getAxis("SelectHorizontal") * deltaTime * 60;
5   float verticalAxis = PlayerInputs.getAxis("SelectVertical") * deltaTime * 60;
6
7   if (horizontalAxis != 0 || verticalAxis != 0) {
8     // otočit předmětem ve scéně
9     // uložení události pro kalendář
10    recorder.addEvent(new ObjectControllerAxisEvent(ghost, horizontalAxis,
          verticalAxis));
11
12    // provedení události na hráčovi
13    applyControllerAxisEvent(horizontalAxis, verticalAxis);
14  }
15
16  if (PlayerInputs.getButtonDown("PickUp")) { //sebrat předmět
17    recorder.addEvent(new InventoryEvent(ghost, PlayerInventoryEvent.PickUp));
18    applyInventoryEvent(PlayerInventoryEvent.PickUp);
19  }
20
21  if (PlayerInputs.getButtonDown("Use")) { // použít předmět
22    recorder.addEvent(new InventoryEvent(ghost, PlayerInventoryEvent.UseDown));
23    applyInventoryEvent(PlayerInventoryEvent.UseDown);
24  }
25
26  if (PlayerInputs.getButtonDown("Jump")) { // vyskočit nebo se postavit
27    recorder.addEvent(new SpecialMovementEvent(ghost, PlayerSpecialMovement.Jump));
28    applySpecialMovementEvent(PlayerSpecialMovement.Jump);
29  }
30
31  if (PlayerInputs.getButtonDown("Flashlight")) { // zapnout nebo vypnout baterku
32    recorder.addEvent(new FlashlightEvent(ghost));
33    switchFlashlight();
34  }
```

Výpis 3.9: Příklady vytvoření a uložení některých událostí.

Délka fáze nahrávání je časově omezena. Fáze nahrávání se ukončí buď automaticky, když vyprší čas nebo když hráč chce ukončit nahrávání a vrátit se v čase. Po fázi nahrávání se automaticky spustí fáze přehrávání.

3.12.2 Fáze přehrávání

Hráč se vrátí v čase s předmětem na místo, kde byl při spuštění fáze nahrávání. Bude ve stavu z konce nahrávání s předmětem stejného typu, který měl na konci nahrávání. Na stejné místo kde je hráč se aktivuje Duch ve stavu, ve kterém byl hráč při spuštění nahrávání. V kalendáři se následně aktivuje přehrávání událostí. Dokud je aktivní fáze přehrávání, nelze spustit novou fázi nahrávání.

Kalendář událostí

Implementoval jsem diskrétní kalendář událostí [8]. Hlavní metoda kalendáře (provádění událostí v čase) je popsána pomocí pseudokódu (viz. 3). Kalendář provádí události v pořadí, v jakém byly vytvořeny. Kalendář slouží pro pohyb Ducha a volání akcí na Duchovi. Každý snímek aktualizuje čas přehrávání a posune Ducha. Kalendář může být pozastaven v důsledku překážky před Duchem. Překážkou mohou být například zavřené dveře, kterými předtím hráč prošel ve fázi nahrávání. U některých událostí záleží, zda byly úspěšně provedeny. Pokud se aktuální událost nepodařilo provést, kalendář nepřejde na další událost a pokusí se zopakovat aktuální událost o snímek později.

Kalendář neobsahuje ukončení. V CSharpSkriptu existuje implementace **yield**. Čekání jeden snímek je prováděno přes příkaz „yield return null“. Každý snímek hry, kdy je kalendář aktivní, engine Unity do kalendáře vstoupí, provede 0 až N událostí a přeš yield kód zas opustí. Ukončení provádění akcí kalendáře se provádí ukončením vstupování engine zpět do metody.

3.12.3 Dočasné objekty

Ve scéně jsou dva druhy objektů. Trvalé a dočasné. Trvalé objekty se vytvoří při spuštění scény a existují až do jejího skončení. Například hráč, Duch, nepřítel, stěny, laser, dveře a další. Některé se mohou vypínat (např. umře nepřítel), ale mají po celou dobu alokovanou paměť. Dočasné objekty jsou v průběhu hry vytvářeny a po čase odstraněny. Dočasným objektem je například letící kulka nebo efekty úderu předmětu. Obvykle ve hrách, když např. kulka narazí do objektu, vytvoří se efekt dopadu kulky a kulka se odstraní (dealokuje se i z paměti).
V důsledku mechaniky cestování v čase vzniká problém s dočasný objekty. Objekty, které by se měly odstranit nesmí být odstraněny, ale pouze vypnuty. Až se hráč vrátí v čase, vypnuté dočasné objekty se opět zapnou a vrátí se do uloženého stavu.

Když např. hráč vystřelí ze zbraně, vytvoří se kulka, o snímek později hráč zapne nahrávání a hráč se časem vrátí v čase do bodu, když nahrávání spustil. Kulka opět musí být zpět před hráčem.

Dočasné objekty nejsou odstraňovány, když mají být odstraněny nebo když jim vyprší životnost. Objekty jsou odstraňovány podle toho, v jaké fázi hry byly vytvořeny a v jaké fázi se chtěly odstranit. Objekty se řídí pravidly:

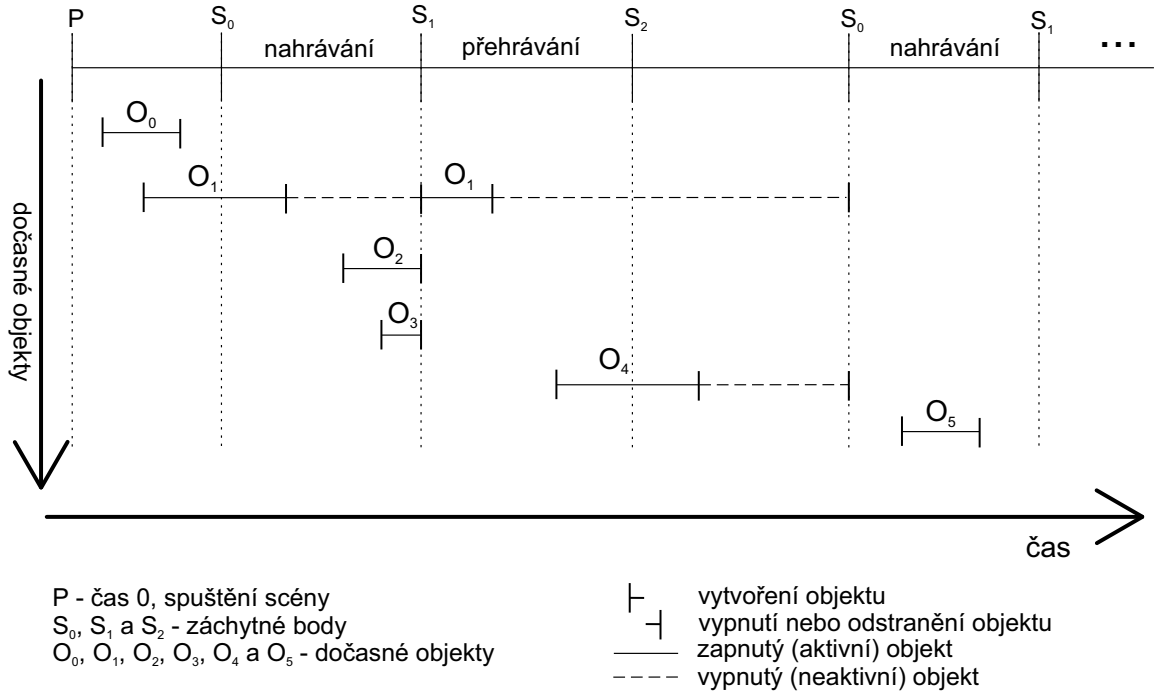
- Objekt se může odstranit, pokud byl vytvořen ve stejné fázi, ve které má být odstraněn.

```
Function runCalendar() /* spusť kalendář událostí                    */
    nastav první událost jako aktuální;
    čas do události = 1;
    /* žádná událost ani existovat nemusí, ale je potřeba, aby kalendář
       pořád pohyboval s Duchem                                        */
    while true do
        if kalendář je pozastaven then
            /* Duch nemůže někam projít                                */
            počkej jeden snímek;
            continue;
        end
        if aktuální událost existuje then
            zjisti čas do události;
        end
        if čas do události > 0 then
            počkej jeden snímek;
            inkrementuj čas přehrávání o čas mezi snímky;
            posuň Ducha;
        else
            zapamatuj si čas přehrávání;
            nastav čas přehrávání na čas události;
            posuň Ducha na pozici;
            /* Duch musí být na pozici, kde byl hráč, když událost byla
               nahrána                                                 */
            proveď aktuální událost;
            nastav zpět čas přehrávání na zapamatovanou hodnotu;
            if událost se provedla then
                aktuální událostí je událost následující čas do události = 1;
            else
                počkej jeden snímek;
                /* událost se nepodařilo provést, kalendář se na snímek
                   pozastaví a zopakuje stejnou akci v dalším snímku   */
            end
        end
    end
end
```

Algorithm 3: Pseudokód diskrétního kalendáře události. Aktivuje jednotlivé události v průběhu času – ovládá Ducha.

- Pokud objekt byl vytvořen v jiné fázi, než ve které má být odstraněn, objekt se pouze vypne.
- Na konci fáze nahrávání jsou vždy smazány všechny objekty, které byly ve fázi nahrávání vytvořeny. Nezáleží, že ještě chtěly existovat, odstraní se.
- Objekty, které jsou vypnuty a čekají na odstranění jsou odstraněny vždy se zahájením fáze nahrávání.

Příklad s dočasnými objekty je na obrázku 3.26



Obrázek 3.26: Na obrázku je zobrazena životnost dočasných objektů v průběhu hry. Objekty jsou vytváření, vypínány a odstraňovány v průběhu času napříč fázemi:

- Objekt O_0 byl vytvořen ve fázi nečinnosti. Ve stejné fázi požádal o odstranění a byl odstraněn. Odstranění bylo dovoleno, protože byl vytvořen ve stejné fázi jako ve které se chtěl odstranit.
- Objekt O_1 byl vytvořen ve fázi nečinnosti. Ve fázi nahrávání požádal o odstranění, ale nebyl odstraněn, pouze vypnut. Ve fázi přehrávání byl znovu aktivován a znovu se pouze vypnul místo odstranění. Objekt byl odstraněn až při zahájení nového nahrávání.
- Objekty O_2 a O_3 byly vytvořeny ve fázi nahrávání. Oba objekty na konci fáze nahrávání byly odstraněny, protože před fází nahrávání ještě neexistovaly.
- Objekt O_4 byl vytvořen ve fázi přehrávání. Ve fázi nečinnosti požádal o odstranění, ale byl pouze vypnut, pro případ, že by se hráč chtěl vrátit do záchytného bodu S_2. Nastala znovu fáze nahrávání a vypnutý objekt O_4 byl odstraněn.
- Objekt O_5 byl vytvořen ve fázi nahrávání a ve stejné fázi požádal o odstranění a byl odstraněn. Princip stejný jako s O_0.

3.12.4 Body návratu

Při cestování v čase a způsobování časových paradoxů může nastat problém pro hráče. Hráč si nesprávně rozmyslí akce a pak Duch nebude moct projít například dveřmi, kterými hráč při nahrávání prošel. Pokud hráč nemá možnost, jak tyto dveře pro Ducha otevřít, Duch zůstane čekat na pořád. Fáze přehrávání tak nikdy neskončí, nahrávací oblast nelze opustit a nové nahrávání taky nelze provést. V takovém případě se hráč musí vrátit do některého z uložených stavů hry. Výběr uložených stavů je v herním menu v podmenu „Záchytný bod".

V menu si hráč může vybrat, do jakého místa se chce vrátit, možnosti jsou:

- Znovu načíst scénu – scéna se znovu načte a hráč bude zpět na začátku scény.
- Záchytný bod – hráč se vrátí na poslední záchytný bod (poslední protnutý záchytný bod).
- Před bod nahrávání – hra bude ve stavu, než hráč spustil nahrávání. Nahrávání bude vypnuté. Vhodné použít, když se Duch zasekne o překážku v důsledku časového paradoxu a víme, že překážky se nezbavíme.
- Znovu nahrávání – hra bude ve stavu, když hráč spustil nahrávání. Nahrávání bude zapnuté.
- Znovu přehrávání – hra bude ve stavu, když se spustilo přehrávání. Přehrávání bude zapnuté (bude aktivní Duch). Vhodné použít, když jsme dobře nahráli akce, ale při přehrávání jsme udělali chybu. Nemusíme nahrávání znovu provádět, spustí se jen přehrávání.
- Na konec přehrávání – hra bude ve stavu, když se ukončilo přehrávání a Duch zmizel ze scény. Hráč mohl zvládnout nahrávání i přehrávání akcí, ale pak něco pokazil a nechce procházet stejnou část znovu.

Kapitola 4

Uživatelské testování

Hra je druh softwaru, který musí být testován i na skutečných uživatelích a od uživatelů musí být získána zpětná odezva. Zpětnou odezvu lze získat například pomocí dotazníků, ale to podle mě není dobrý způsob pro získávání zpětné odezvy u hry. Například zjišťovat na uživateli, zda ho hra baví, nelze dotazníkem.

Dotazník je jedna z nejběžnějších forem získávání zpětné vazby od uživatelů. Pročítal jsem různé bakalářské práce s herní tématikou a velmi často končily dotazníkem [20]. To by nebylo přímo nic špatného. Z dotazníku se někdy dají získat užitečné informace. Například jak moc se hráči líbil grafický styl pomocí číselné stupnice nebo kolik času mu zabralo než prošel jednotlivé scény. Ale většina dotazníků na získávání odezvy od uživatelů testující hru (hlavně dotazníky vytvořeny studenty) jsou podle mě špatně. Výsledky z dotazníků jsou podle mě často zavádějící a nepoužitelné.

Uživatelům jsem taky dával běžné otázky jako jsou:

- „Líbila se ti hra?“
  „Ano.“
- „Bavila tě hra?“
  „Ano.“
- „Naučila tě hra něco nového?“
  „Ano.“
- „Bylo pro vás náročné ovládání hry?“
  „S ovládáním ne, spíš si uvědomit, co dělat dál.“

Ale když dáme uživateli hru zahrát a pak se ho zeptáme, zda se mu hra líbila nebo zda ho bavila, tak už ze slušnosti a aby nás neurazil, odpoví že „Ano“.

V jedné práci jsem narazil i na otázku v dotazníku „V čem vás hra mile překvapila?“. Odpovědí bylo „mini hrami“. Zde bych odpovědi v dotazníku považoval možná i za smyšlené. Když si někdo jde zahrát hru o určitých herních mechanikách nebo příběhu, málo kdy ho zajímají mini hry. A pokud hra obsahuje jen a pouze mini hry, tak vás mini hry nemohou překvapit. To, že si bude moct zahrát Tetris nebo piškvorky na displeji někde ve hře obvykle nikoho nezajímá. Pokud by někdo chtěl hrát piškvorky, spustil by si hru piškvorky. Pokud někdo chce hrát vaši hru, tak ho zajímají mechaniky, které jsou ve hře použity. Když dáváte uživateli hru zahrát (otestovat), je potřeba aby ho ve hře něco zaujalo a chtěl hru hrát. Obvykle to jsou herní mechaniky nebo grafická podoba hry.

Betatesting jsem prováděl na živo na uživatelích. Je to podle mě jeden z nejlepších způsobů testování hry, když já jako tvůrce aplikace mohu sledovat uživatele, jak aplikaci používá a bez toho, abych ho výrazněji ovlivňoval. Jako tvůrce mám určitou představu, jak uživatel má hru hrát. Pokud uživatel narazí na příliš obtížnou překážku a neumí si s ní poradit nebo používá předmět špatně pořád a pořád dokola, tak je to chyba, kterou bych měl opravit. Z dotazníku nepoznáte, že se hráč nedokáže vyznat ve scéně, že prošel kolem předmětu, ale nesebral ho. Nemusel totiž ani vědět, že to, kolem čeho prošel je předmět, který lze sebrat. A jak poznat, zda se uživateli hra líbí a baví ho? Sledovat jeho emoce a jeho zapálení do hry. Když hráč vyřeší hádanku nebo porazí nepřítele, má být z toho nadšen či potěšen a chtít hru hrát dál. Pokud je hráč ale naopak znuděn a těší se, až odejdete, tak ho ta hra asi nebaví, i když vám tvrdí, že hra byla dobrá.

Testování probíhalo na různých uživatelích. Někteří, co hrávají logické hry běžně, jiní co hrávají spíše jen hry kde se střílí, a i na jiných co nehrají hry skoro vůbec. Mechaniky jako je chůze s postavou, to dokázal zvládat každý. Je to jedna z nejběžnějších herních mechanik, co obsahuje hodně her a lidi jsou na ni zvyklí. Vyřešit hádanky v počátečních scénách dokázali taky zvládnout skoro všichni, ale problémy začali mít postupně lidé co běžně nehrávají logické hry. Vyřešit hádanky s cestováním zpět v čase se ukázaly jako nejnáročnější. Uživatelé v průběhu hry i zapomínali na herní mechaniky, které použili v jednodušších úrovních a v úrovních, kde je měli kombinovat měli problémy.

Při testování na uživatelích jsem objevoval a následně opravoval mnoho problémů a chyb. Základní věci, jako že se uživateli podařilo zablokovat inventář, uvíznout s postavou v geometrii scény nebo naopak, že určitou geometrií scény lze omylem projít. Takové problémy vývojář běžně přehlídne, protože hru testuje často pořád stejně, ale jiný uživatel hru hraje jinak a projeví se problém, který se u vývojáře neprojevil. Často se stávalo, že uživatel nevěděl, co má dělat, kam má něco položit nebo kudy vlastně přišel. Ztráta orientace byla běžná, protože textury lokací jsou všechny podobné a modely taky. Tyto problémy jsem řešil značkami a doprovodným textem ve scénách. Když se hráč poprvé setká s nějakým typem hádanky nebo herní mechaniky, je u ní napsán krátký doprovodný text, co má udělat. Na různých místech jsem přidal značky, aby hráč dokázal rozpoznat kudy přišel a kam má jít. Na určitá místa další jiné typy značek naznačující, že toto místo je něčím důležité a měl by tu něco udělat. Doprovodný text u hádanek pomohl. Často hráči napověděl, co má udělat nebo mu připomněl, jak některé herní mechaniky fungují.

Prvotním cílem bylo vytvořit hru, aby měla herní dobu kolem 30 minut. Když už víte, jak vyřešit hádanky a porazit nepřítele, hru dokážete projít i pod 20 minut. A však uživatelům, co hru hráli poprvé, trvalo více jak 2 hodiny, než se dostali do poslední úrovně. Přijít na to, jak vyřešit hádanky ve složitějších scénách jim trvalo déle, než jsem předpokládal.

Z testování na uživatelích jsem zjistil, že jsem vytvořil hodně náročné herní mechaniky. Uživatelé, co logické hry nehrávají, bez pomoci nebyli schopni hru dohrát. Některým uživatelům jsem musel i opakovaně poradit. Hlavní problém byl, že si neuvědomovali, jak herní mechaniky přesně fungují a byli z toho zmateni a frustrovaní. Když jsem jim přesně a podrobně vysvětlil, jak herní mechaniky fungují a co bude důsledkem jejích akcí, hru začali chápat, byli ji schopni hrát a líbila se jim.

Kapitola 5

Závěr

Cílem práce bylo navrhnout a implementovat inovativní 3D hru v egine Unity. Nevytvořit jen prototyp, ale vytvořit fungující hru. Musel jsem navrhnout a vytvořit jednotlivé úrovně. Lehčí úrovně, které demonstrují jednotlivé herní mechaniky a těžší, kde hráč musí přemýšlet, kombinovat herní mechaniky a využit je ve svůj prospěch. Hra je ojedinělá především mechanikami schopnostmi hráče zpomalovat čas a cestovat zpět v čase.

Pro tvorbu hry jsem použil různé nástroje zabudované v editoru Unity. Vytvořil jsem postavu hráče, animace, inventář, předměty do inventáře, HUD, menu, nepřítele, vizuální efekty a vymodeloval úrovně). Použil jsem pro hru jak Unity komponenty, tak k nim jsem vytvořil mnoho dalších, které by se daly použít i v jiných hrách založených na Unity (pohyb postav, životy, správa kolizí a *ASM*). Engine a vestavěné komponenty jsem popisoval jen výjimečně, většinou jen když jsem chtěl poukázat na jejich problémy a nevýhody. Ukázal jsem některé výhody mých komponent oproti komponentám z Unity (např. navigace vojáka a kolize). Ukázal jsem několik problémů engine či her obecně a jak je vyřešit (kamery, vrstvy a kolize).

Hra je rozdělena do 10 scén (9 úrovní a menu). Některé úrovně se dělí ještě na menší části. Pro některé úrovně (kombinaci hádanek) existuje více způsobů řešení. Z uživatelského testování jsem zjistil, že některé mnou vymyšlené herní mechaniky jsou až tak ojedinělé, že je vůbec uživatelé neznali a pro řešení hádanek byly hodně náročné. Mechaniky se prokázaly až tak složité, ale zároveň dobré, že by se hru dalo rozdělit a vytvořit z ní minimálně dvě jiné. Jednu, kde by hráč pouze pomocí zpomalování času porážel nepřítele. V další hře pomocí cestování v čase řešil hádanky.

Rozšíření do budoucna:

Hru bych rád upravil, aby ji bylo možné umístit do obchodu Steam. Ve hře chybí detailní grafika. Pro hru jsem vytvořil postavu a animace, které ale slouží spíš jen jako prototyp. Bylo by vhodné vytvořit detailní postavu a animace speciálně pro tuto hru. Vytvoření postavy bych přenechal profesionálnímu grafikovi a vlastní animace vytvořil v některém pokročilém *MoCap* systému. Např. pomocí *MoCap* systému *Perception Neuron* [5]. Jeho základem je oblek obsahující senzory (gyroskop, akcelerometr a magnetometr).

Obohatit hru o další zvuky. Bylo by vhodné přidat např. hlášky nepřítele a zvuky okolí. Vytvořit další úrovně a předměty do inventáře. Přidat různé druhy prostředí a modelů. Případně vytvořit editor na tvorbu úrovní. Hráč by mohl vytvářet a sdílet své úrovně.

Literatura

[1] Eberly, D. H.: *3D game engine design*. Morgan Kaufmann Publishers, 2001, ISBN 978-0122290633.

[2] Gabriel Williams a Karl Henkel: *ProBuilder*. 2018, [Online; navštíveno 25.9.2017]. URL `http://www.procore3d.com/`

[3] Holthe, O.: *PhysX Programming*. Grid Publishing, 2013, ISBN 8293179112.

[4] NICHOLAS J.J. SMITH: *Time Travel*. 2018, [Online; navštíveno 14.4.2018]. URL `https://plato.stanford.edu/entries/time-travel/index.html#CauLoo`

[5] Noitom Ltd.: *Perception Neuron*. 2018, [Online; navštíveno 07.03.2018]. URL `https://neuronmocap.com/`

[6] Okita, A.: *Learning C# Programming with Unity 3D*. CRC Press, 2014, ISBN 978-1-4665-8652-9.

[7] Pedro Nogueira: *Motion Capture Fundamentals*. 2011, [Online; navštíveno 07.05.2018]. URL `https://paginas.fe.up.pt/~prodei/dsie12/papers/paper_7.pdf`

[8] Petr Peringer: *Modelování a simulace*. Studijní opora, VUT Brno, Fakulta informačních technologií, 2012, [Online; navštíveno 21.09.2017].

[9] Schell, J.: *The art of game design: a book of lenses*. Morgan Kaufmann Publishers, 2008, ISBN 978-0-12-369496-6.

[10] Unity Technologies: *Animation System Overview*. 2018, [Online; navštíveno 15.11.2017]. URL `https://docs.unity3d.com/Manual/AnimationOverview.html`

[11] Unity Technologies: *Audio Mixer*. 2018, [Online; navštíveno 15.04.2018]. URL `https://docs.unity3d.com/Manual/AudioMixer.html`

[12] Unity Technologies: *Building a NavMesh*. 2018, [Online; navštíveno 20.01.2018]. URL `https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html`

[13] Unity Technologies: *GUI system*. 2018, [Online; navštíveno 28.11.2017]. URL `https://docs.unity3d.com/Manual/GUIReference.html`

[14] Unity Technologies: *Layers*. 2018, [Online; navštíveno 15.04.2018]. URL `https://docs.unity3d.com/Manual/Layers.html`

[15] Unity Technologies: *NavMesh Agent*. 2018, [Online; navštíveno 20.01.2018].
URL https://docs.unity3d.com/Manual/class-NavMeshAgent.html

[16] Unity Technologies: *UI system*. 2018, [Online; navštíveno 28.11.2017].
URL https://docs.unity3d.com/Manual/UISystem.html

[17] Unity Technologies: *Unity Asset Store*. 2018, [Online; navštíveno 27.10.2017].
URL https://assetstore.unity.com/

[18] Unity Technologies: *Unity Scripting Reference*. 2018, [Online; navštíveno 27.10.2017].
URL https://docs.unity3d.com/ScriptReference/

[19] Unity Technologies: *Unity User Manual*. 2018, [Online; navštíveno 27.10.2017].
URL https://docs.unity3d.com/Manual/index.html

[20] Velek, A.: *Výukově orientovaná 3D hra v prostředí Českého Krumlova*. Bakalářská práce, Jihočeská univerzita v Českých Budějovicích, Fakulta Pedagogická, 2013.
URL https://theses.cz/id/frg497/
Vukov_orientovan_3D_hra_v_prosted_eskho_Krumlova_Velek.pdf

[21] Wasson, T.: *Rigidbodies experience huge changes in velocity if they are colliding when I change Time.timeScale*. 2014, [Online; navštíveno 12.11.2017].
URL
https://forum.unity.com/threads/rigidbodies-experience-huge-changes-in-velocity-if-they-are-colliding-when-i-change-time-timescale.275707/

Příloha A

Obsah DVD

- project – Projekt pro Unity se všemi zdrojovými soubory aplikace.
- bin – Přeložená aplikace pro platformy Windows x86 a Linux x86.
- doc – Technická zpráva ve formátu PDF a zdrojové texty pro prostředí LaTeX.
- video – Demonstrační video.

Příloha B

Manual

Tato část přílohy obsahuje informace pro překlad aplikace, spuštění a ovládání.

Potřebný software

Pro překlad je potřeba Unity Beta 2018. Projekt je přepnut do verze 2018.1.0b12. Pro otevření musí být projekt uložen v adresáři s povoleným zápisem. Pro překlad není třeba v Unity nic nastavovat.

Spuštění

Hru lze hrát ve fullscreen režimu nebo v okně. V různých poměrech obrazu (např. 16:9 nebo 4:3). Po spuštění aplikace se zobrazí dialogové okno. Jedná se o okno automaticky vytvořené v Unity. V záložce „Input“ lze přemapovat ovládání. Tlačítkem „Play!“ se spustí hra. Po spuštění hry se zobrazí hlavní menu. V podmenu „Hrát“ tlačítkem „Nová hra“ se provede načtení první úrovně. Přes podmenu „Načíst“ lze načíst libovolnou úroveň.

Ovládání

- W, A, S, D – pohyb dopředu, doleva, dozadu, doprava
- Left Shift – běh
- E – interakce (sebrání předmětu nebo zmáčknutí tlačítka ve scéně)
- Q – upustit předmět
- LMB – použit předmět
- RMB – hodit předmět
- Arrows – otáčení objektu (např. zrcadla)
- Space – výskok nebo stoupnout
- C – skrčit nebo stoupnout
- Tab – zpomalit nebo zrychlit čas
- V – zapnout nebo vypnout svítilnu

- R – spustit nahrávání nebo ukončit nahrávání
- T – opakovat přehrávání
- F – stejné jak klávesa E. V režimu nahrávání událostí je s čekáním. Duch bude čekat na místě, dokud mu např. nebude podán předmět.
- Esc – vyvolá herní menu nebo návrat z podmenu